

# 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.



# Reporting Errors and Omissions

This book is a work-in-progress produced by and for the MEGA65 community. The version of this edition is:

```
commit 8bd31f50960f5d12aa848886b44ab140b1c8f689
Date:   Thu Jul 16 19:56:28 2020 +0930
```

We want this book to be the best that it possibly can. So if you see any errors, or find anything that is missing, or that you would like more information on, please report them using the MEGA65 User Guide issue tracker:

<https://github.com/mega65/mega65-user-guide/issues>

You can also check there to see if anyone else has reported a similar problem, while you wait for this book to be updated.

Finally, you can always download the latest version of this book from:

<https://github.com/mega65/mega65-user-guide>



# 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.

July 16, 2020

# Contents

<b>1</b>	<b>Introduction</b>	<b>v</b>
<b>I</b>	<b>GETTING TO KNOW YOUR MEGA65</b>	<b>1</b>
<b>2</b>	<b>SETUP</b>	<b>3</b>
	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 CURSOR . . . . .	10
<b>3</b>	<b>GETTING STARTED</b>	<b>11</b>
	Keyboard . . . . .	13
	Command Keys . . . . .	13
	RETURN . . . . .	13
	SHIFT . . . . .	13

SHIFT LOCK . . . . .	13
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 . . . . .	17
Editor Functionality . . . . .	19
<b>4 Configuring your MEGA65</b>	<b>21</b>
Preparing for First Use: Formatting SD Cards . . . . .	23
Installing ROM and Other Support Files . . . . .	26
Configuring your MEGA65 . . . . .	27
Input Devices . . . . .	31
Chipset . . . . .	32
Video . . . . .	33
Audio . . . . .	34
Network . . . . .	35
<b>5 Cores and Flashing</b>	<b>37</b>
What are cores, and why does it matter? . . . . .	39
Selecting a core . . . . .	39

Installing an upgrade core for the MEGA65 . . . . .	41
Installing other cores . . . . .	44
Creating cores for the MEGA65 . . . . .	45
Replacing the factory core in slot 0 . . . . .	45
Understanding The Core Booting Process . . . . .	45
<b>II FIRST STEPS IN CODING</b>	<b>47</b>
<b>6 How Computers Work</b>	<b>49</b>
Computers are stupid. Really stupid . . . . .	51
Making an Egg Cup Computer . . . . .	51
<b>7 Getting Started in BASIC</b>	<b>53</b>
Your first BASIC programmes . . . . .	55
Exercises to try . . . . .	65
First steps with text and numbers . . . . .	65
Exercises to try . . . . .	81
Making simple decisions . . . . .	82
Exercises to try . . . . .	96
Random numbers and chance . . . . .	97
Exercises to try . . . . .	102
<b>8 Text Processing</b>	<b>105</b>
Characters and Strings . . . . .	107
String Literals . . . . .	109
String Variables . . . . .	109
String Statements . . . . .	111
Simple Formatting . . . . .	112
Suppressing New Lines . . . . .	112

Automatic Tab Stops . . . . .	112
Tabs Stops and Spacing . . . . .	112
Sample Programs . . . . .	113
Palindromes . . . . .	113
Simple Ciphers . . . . .	113
<b>9 C64, C65 and MEGA65 Modes</b>	<b>121</b>
Switching Modes from BASIC . . . . .	123
From C65 to C64 Mode . . . . .	123
From C64 to C65 Mode . . . . .	123
Entering Machine Code Monitor Mode . . . . .	124
The KEY Register . . . . .	124
Un-hiding C65 Extra Registers . . . . .	124
Re-hiding C65/MEGA65 Extra Registers . . . . .	125
Un-hiding MEGA65 Extra Registers . . . . .	126
Traps to watch out for . . . . .	127
Accessing the MEGA65's Extra Memory from BASIC 10 in C65 Mode . . . . .	127
The MAP Instruction . . . . .	128
<b>III SOUND AND GRAPHICS</b>	<b>129</b>
<b>IV HARDWARE</b>	<b>131</b>
<b>10 Using a Nexys4DDR as a MEGA65</b>	<b>133</b>
Building your own MEGA65 Compatible Computer . . . . .	135
Power, Jumpers, Switches and Buttons . . . . .	136
Micro-USB Power . . . . .	136
External Power Supply . . . . .	136
Other Jumpers and Switches . . . . .	137

Connections and Peripherals . . . . .	138
Onboard buttons . . . . .	138
Keyboard . . . . .	139
Preparing microSDHC card . . . . .	140
Useful Tips . . . . .	141

## V APPENDICES 143

### A ACCESSORIES 147

### B BASIC 10 Command Reference 149

Format of Commands, Functions and Operators . . . . .	151
Commands, Functions and Operators . . . . .	153
ABS . . . . .	154
AND . . . . .	155
APPEND . . . . .	156
ASC . . . . .	157
ATN . . . . .	158
AUTO . . . . .	159
BACKGROUND . . . . .	160
BACKUP . . . . .	161
BANK . . . . .	162
BEGIN . . . . .	163
BEND . . . . .	164
BLOAD . . . . .	165
BOOT . . . . .	166
BORDER . . . . .	167
BOX . . . . .	168
BSAVE . . . . .	169

BUMP	170
BVERIFY	171
CATALOG	172
CHANGE	173
CHAR	174
CHR\$	175
CIRCLE	176
CLOSE	177
CLR	178
CMD	179
COLLECT	180
COLLISION	181
COLOR	182
CONCAT	183
CONT	184
COPY	185
COS	186
CURSOR	187
DATA	188
DCLEAR	189
DCLOSE	190
DEC	191
DEF FN	192
DELETE	193
DIM	194
DIR	195
DISK	196
DLOAD	197

DMA	198
DMODE	199
DO	200
DOPEN	201
DPAT	203
DSAVE	204
DVERIFY	205
EL	206
ELLIPSE	207
ELSE	208
END	209
ENVELOPE	210
ERASE	211
ER	212
ERR\$	213
EXIT	214
EXP	215
FAST	216
FILTER	217
FIND	218
FN	219
FOR	220
FOREGROUND	221
FRE	222
GET	223
GET#	224
GETKEY	225
GO64	226

GOSUB .....	227
GOTO .....	228
GRAPHIC .....	229
HEADER .....	230
HELP .....	231
HEX\$ .....	232
HIGHLIGHT .....	233
IF .....	234
INPUT .....	235
INPUT# .....	236
INSTR .....	237
INT .....	238
JOY .....	239
KEY .....	240
LEFT\$ .....	241
LEN .....	242
LET .....	243
LINE .....	244
LIST .....	245
LOAD .....	246
LOCATE .....	247
LOG .....	248
LOOP .....	249
LPEN .....	250
MID\$ .....	251
MOD .....	252
MONITOR .....	253
MOUSE .....	254

MOVSPR . . . . .	255
NEW . . . . .	256
NEXT . . . . .	257
NOT . . . . .	258
OFF . . . . .	259
ON . . . . .	260
OPEN . . . . .	262
OR . . . . .	263
PAINT . . . . .	264
PALETTE . . . . .	265
PEEK . . . . .	266
PEN . . . . .	267
PLAY . . . . .	268
POINTER . . . . .	270
POKE . . . . .	271
POLYGON . . . . .	272
POS . . . . .	273
POT . . . . .	274
PRINT . . . . .	275
PRINT# . . . . .	276
PRINT USING . . . . .	277
PUDEF . . . . .	279
RCLR . . . . .	280
RDOT . . . . .	281
READ . . . . .	282
RECORD . . . . .	283
REM . . . . .	284
RENAME . . . . .	285

RENUMBER	286
RESTORE	287
RESUME	288
RETURN	289
RGR	290
RIGHT\$	291
RMOUSE	292
RND	293
RREG	294
RSPCOLOR	295
RSPPPOS	296
RSprite	297
RUN	298
SAVE	299
SCNCLR	300
SCRATCH	301
SCREEN	302
SET	303
SGN	304
SIN	305
SLEEP	306
SLOW	307
SOUND	308
SPC	309
SPRCOLOR	310
SPRITE	311
SPRSAV	312
SQR	313

STEP . . . . .	314
STOP . . . . .	315
STR\$ . . . . .	316
SYS . . . . .	317
TAB . . . . .	319
TAN . . . . .	320
TEMPO . . . . .	321
THEN . . . . .	322
TO . . . . .	323
TRAP . . . . .	324
TROFF . . . . .	325
TRON . . . . .	326
TYPE . . . . .	327
UNTIL . . . . .	328
USING . . . . .	329
USR . . . . .	331
VAL . . . . .	332
VERIFY . . . . .	333
VOL . . . . .	334
WAIT . . . . .	335
WHILE . . . . .	336
WINDOW . . . . .	337
XOR . . . . .	338
<b>C Special Keyboard Controls and Sequences</b>	<b>339</b>
PETSCII Codes and CHR\$ . . . . .	341
Control codes . . . . .	343
Shifted codes . . . . .	345
Escape Sequences . . . . .	346

<b>D The MEGA65 Keyboard</b>	<b>349</b>
Hardware Accelerated Keyboard Scanning . . . . .	351
Latin-1 Keyboard Map . . . . .	352
Keyboard Theory of Operation . . . . .	352
C65 Keyboard Matrix . . . . .	352
Synthetic Key Events . . . . .	353
Keyboard LED Control . . . . .	354
Native Keyboard Matrix . . . . .	355
<b>E Decimal, Binary and Hexadecimal</b>	<b>359</b>
Numbers . . . . .	361
Notations and Bases . . . . .	362
Decimal . . . . .	364
Binary . . . . .	366
Hexadecimal . . . . .	368
Operations . . . . .	370
Counting . . . . .	370
Arithmetic . . . . .	372
Logic Gates . . . . .	374
Signed and Unsigned Numbers . . . . .	376
Bit-wise Logical Operators . . . . .	376
Converting Numbers . . . . .	379
<b>F System Memory Map</b>	<b>385</b>
Introduction . . . . .	387
MEGA65 Native Memory Map . . . . .	388
The First Sixteen 64KB Banks . . . . .	388
Colour RAM . . . . .	389
28-bit Address Space . . . . .	390

\$D000 - \$DFFF IO Personalities . . . . .	392
CPU Memory Banking . . . . .	394
C64/C65 ROM Emulation . . . . .	395
C65 Compatibility ROM Layout . . . . .	396
<b>G 45GS02 Microprocessor</b>	<b>397</b>
Introduction . . . . .	399
Differences to the 6502 . . . . .	399
Supervisor/Hypervisor Privileged Mode . . . . .	399
6502 Illegal Opcodes . . . . .	400
Read-Modify-Write Instruction Bug Compatibility . . . . .	400
Variable CPU Speed . . . . .	401
Slow (1MHz - 3.5MHz) Operation . . . . .	401
Full Speed (40MHz) Instruction Timing . . . . .	402
CPU Speed Fine-Tuning . . . . .	402
Direct Memory Access (DMA) . . . . .	402
Accessing memory between the 64KB and 1MB points . . . . .	403
C64-Style Memory Banking . . . . .	403
VIC-III "ROM" Banking . . . . .	403
VIC-III Display Address Translator . . . . .	404
The MAP instruction . . . . .	404
Direct Memory Access (DMA) Controller . . . . .	405
Flat Memory Access . . . . .	405
Accessing memory beyond the 1MB point . . . . .	405
Using the MAP instruction to access >1MB . . . . .	405
Flat-Memory Access . . . . .	407
Virtual 32-bit Register . . . . .	408
C64 CPU Memory Mapped Registers . . . . .	410
New CPU Memory Mapped Registers . . . . .	410

MEGA65 CPU Maths Unit Registers . . . . .	414
MEGA65 Hypervisor Mode . . . . .	421
Reset . . . . .	421
Entering / Exiting Hypervisor Mode . . . . .	422
Hypervisor Memory Layout . . . . .	423
Hypervisor Virtualisation Control Registers . . . . .	426
Programming for Hypervisor Mode . . . . .	428
<b>H 45GS02 &amp; 6502 Instruction Sets</b>	<b>433</b>
Addressing Modes . . . . .	435
Implied . . . . .	435
Accumulator . . . . .	435
Immediate Mode . . . . .	435
Immediate Word Mode . . . . .	436
Base Page (Zero-Page) Mode . . . . .	436
Base Page (Zero-Page) X Indexed Mode . . . . .	436
Base Page (Zero-Page) Y Indexed Mode . . . . .	437
Base Page (Zero-Page) Z Indexed Mode . . . . .	437
Absolute Mode . . . . .	437
Absolute X Indexed Mode . . . . .	437
Absolute Y Indexed Mode . . . . .	438
Absolute Z Indexed Mode . . . . .	438
Absolute Indirect Mode . . . . .	438
Absolute Indirect X-Indexed Mode . . . . .	438
Base Page Indirect X-Indexed Mode . . . . .	439
Base Page Indirect Y-Indexed Mode . . . . .	439
Base Page Indirect Z-Indexed Mode . . . . .	439
32-bit Base Page Indirect Z-Indexed Mode . . . . .	439
Stack Relative Indirect, Indexed by Y . . . . .	440

Relative Addressing Mode . . . . .	440
Relative Word Addressing Mode . . . . .	440
45GS02 Instruction Set . . . . .	441
Opcode Map . . . . .	442
Instruction Timing . . . . .	443
Addressing Mode Table . . . . .	445
6502 Instruction Set . . . . .	504
Opcode Map . . . . .	505
Instruction Timing . . . . .	506
Addressing Mode Table . . . . .	507
Official And Unintended Instructions . . . . .	508
<b>I F018-Compatible Direct Memory Access (DMA) Controller</b>	<b>551</b>
Audio DMA . . . . .	554
Sample Address Management . . . . .	554
Sample Playback frequency and Volume . . . . .	555
Pure Sine Wave . . . . .	556
Sample playback control . . . . .	556
MEGA65 Enhanced DMA Jobs . . . . .	556
F018 “DMAgic” DMA Controller . . . . .	557
MEGA65 DMA Controller Extensions . . . . .	557
Unimplemented Functionality . . . . .	558
<b>J VIC-IV Video Interface Controller</b>	<b>559</b>
Features . . . . .	561
VIC-II/III/IV Register Access Control . . . . .	562
Detecting VIC-II/III/IV . . . . .	563
Video Output Formats, Timing and Compatibility . . . . .	565
Integrated Marvellous Digital Hookup™(IMDH™) Digital Video Output . . . . .	565

Connecting to Naughty Proprietary Digital Video Standards . . . . .	565
Frame Timing . . . . .	567
Physical and Logical Rasters . . . . .	570
Bad Lines . . . . .	570
Memory Interface . . . . .	571
Relocating Screen Memory . . . . .	571
Relocating Character Generator Data . . . . .	571
Relocating Colour / Attribute RAM . . . . .	572
Relocating Sprite Pointers and Images . . . . .	572
Hot Registers . . . . .	573
New Modes . . . . .	574
Why the new VIC-IV modes are Character and Bitmap modes, not Bitplane modes . . . . .	574
Displaying more than 256 unique characters via "Super-Extended Attribute Mode" . . . . .	575
Using Super-Extended Attribute Mode . . . . .	577
Full-Colour (256 colours per character) Text Mode . . . . .	578
Many-colour (16 colours per character) Text Mode . . . . .	579
Alpha-Blending / Anti-Aliasing . . . . .	579
Flipping Characters . . . . .	579
Variable Width Fonts . . . . .	579
Raster-Rewrite Buffer . . . . .	580
Sprites . . . . .	580
VIC-II/III Sprite Control . . . . .	580
Extended Sprite Image Sets . . . . .	581
Variable Sprite Size . . . . .	581
Variable Sprite Resolution . . . . .	582
Sprite Palette Bank . . . . .	582
Full-Colour Sprite Mode . . . . .	583

VIC-II / C64 Registers . . . . .	585
VIC-III / C65 Registers . . . . .	588
VIC-IV / MEGA65 Specific Registers . . . . .	592
<b>K 6526 Complex Interface Adaptor (CIA) Registers</b>	<b>597</b>
CIA 6526 Registers . . . . .	599
CIA 6526 Hypervisor Registers . . . . .	602
<b>L 4551 UART, GPIO and Utility Controller</b>	<b>605</b>
C65 6551 UART Registers . . . . .	607
4551 General Purpose IO & Miscellaneous Interface Registers . . . . .	608
<b>M 45E100 Fast Ethernet Controller</b>	<b>611</b>
Overview . . . . .	613
Differences to the RR-NET and similar solutions . . . . .	613
Theory of Operation: Receiving Frames . . . . .	614
Accessing the Ethernet Frame Buffers . . . . .	615
Theory of Operation: Sending Frames . . . . .	617
Advanced Features . . . . .	617
Broadcast and Multicast Traffic and Promiscuous Mode . . . . .	617
Debugging and Diagnosis Features . . . . .	618
Memory Mapped Registers . . . . .	619
COMMAND register values . . . . .	620
Example Programs . . . . .	621
<b>N Reference Tables</b>	<b>623</b>
Units of Storage . . . . .	625
Base Conversion . . . . .	626

<b>O Flashing the FPGAs and CPLDs in the MEGA65</b>	<b>631</b>
Warning . . . . .	633
Flashing the Artix 100T main FPGA with XILINX VIVADO . . . . .	634
Flashing the CPLD in the MEGA65's Keyboard with LATTICE DIAMOND . . . . .	640
Flashing the MAX10 FPGA on the MEGA65's Mainboard with INTEL QUARTUS . . . . .	647
<b>P Model Specific Features</b>	<b>653</b>
MEGA65 Desktop Computer, Revision 2 onwards . . . . .	655
MEGApone Handheld, Revisions 1 and 2 . . . . .	656
Nexys4 DDR FPGA Board . . . . .	657
<b>Q Supporters &amp; Donors</b>	<b>659</b>
Organisations . . . . .	661
Volunteers . . . . .	661
Individual Donors . . . . .	661
<b>INDEX</b>	<b>667</b>
<b>VI ELEMENT CATALOGUE</b>	<b>677</b>
Graphic Symbols Font . . . . .	678
Handy Symbols . . . . .	678
Keyboard keys . . . . .	679
Screen Output . . . . .	679
Screen font mapping . . . . .	680
Sprite Grids . . . . .	680
Balloon Sprite Demo . . . . .	680
Multi-Colour Sprite . . . . .	681

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® 65<sup>1</sup> 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™<sup>2</sup> computer.

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

---

<sup>1</sup>Commodore is a trademark of C= Holdings

<sup>2</sup>Commodore 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 you may need:

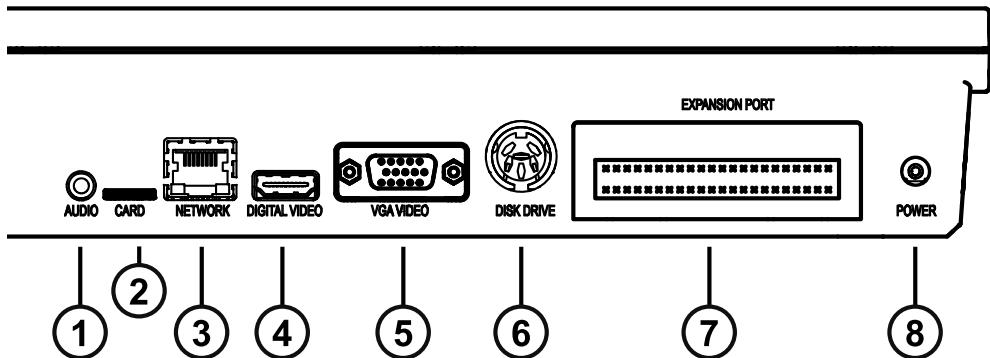
- A television or computer monitor with a VGA or digital video input, that is capable of displaying an image with 800x600 pixel resolution at 50Hz or 60Hz.
- A VGA video cable, or;
- A digital video cable.

These items are not included with the MEGA65.

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

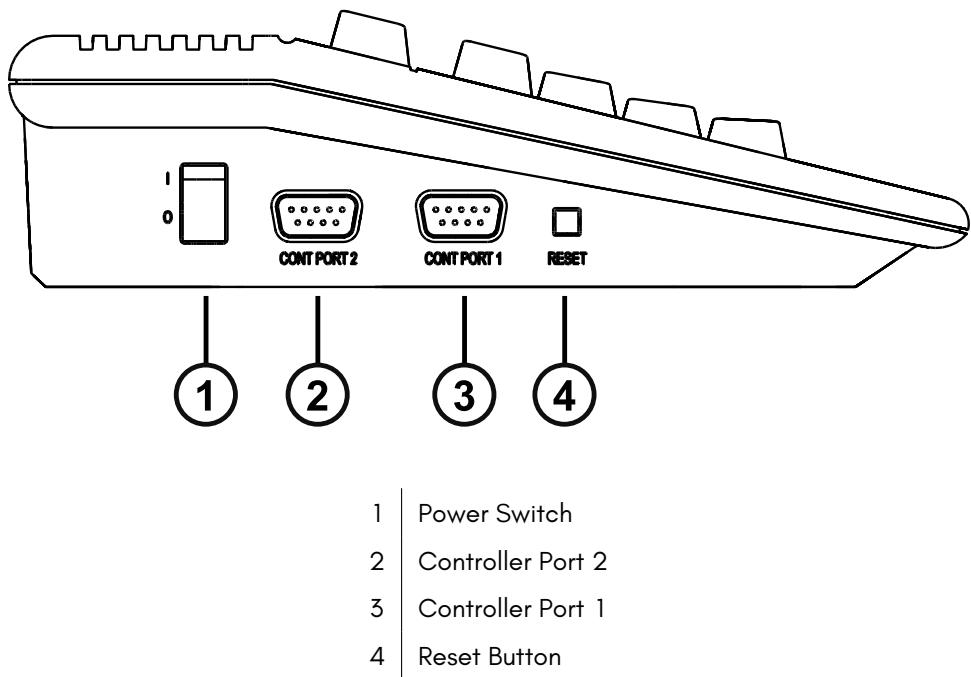
- 3.5mm mini-jack audio cable and suitable speakers or hifi system, so that you can enjoy the sound capabilities of your MEGA65.
- RJ45 Ethernet cable (regular network cable) and a network router or switch. This allows the usage of the high-speed networking capabilities of your MEGA65.

# 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         |

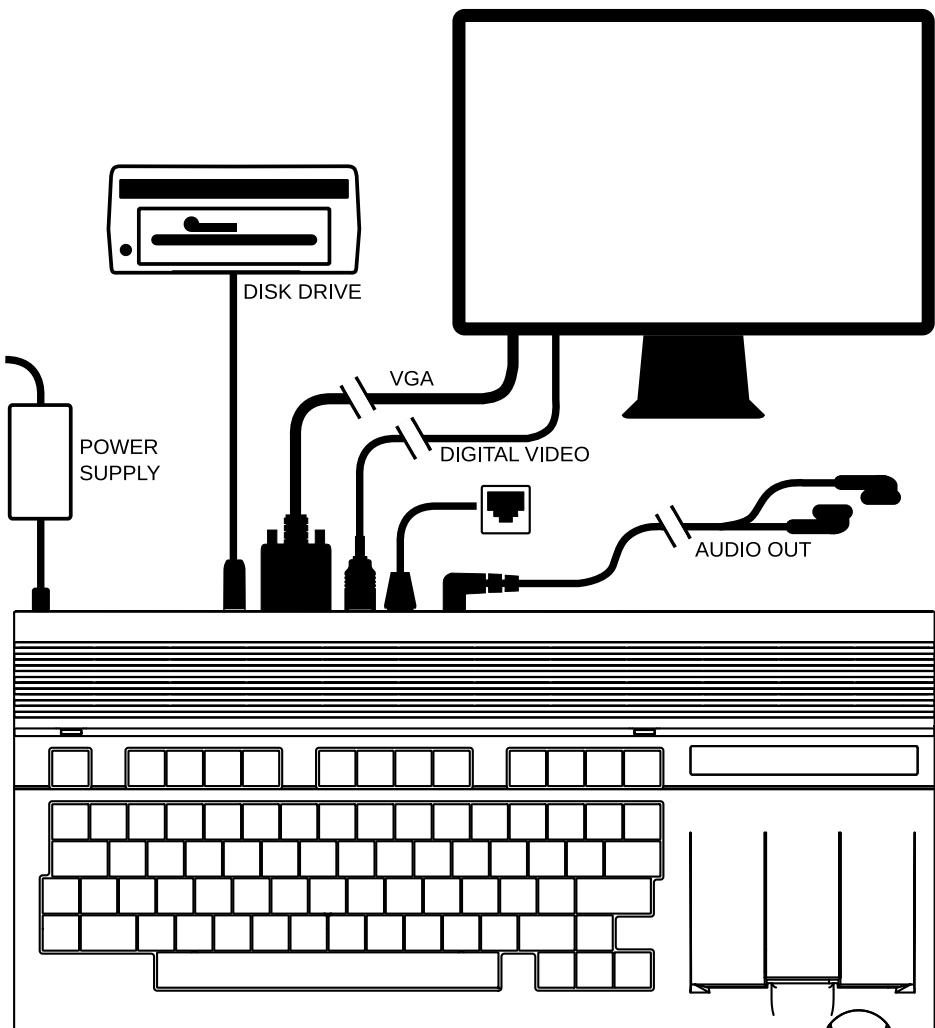
# SIDE CONNECTIONS



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 compatible 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 adaptor.

## 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 adaptor 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**, **M** 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 down **SHIFT** and pressing any key with a graphic symbol on the front produces the right hand symbol on that key. For example, **S** and **J** prints the ☒ character.

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

Finally, holding down 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 **CTRL** and pressing another key allows you to perform Control Functions. For example, holding down **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 down **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**. This combination also resets operating system vectors and re-initialises the screen editor, which makes it a handy combination if the computer has become a little confused.

Programs are able to disable this key combination.

Enter the Freeze Menu by holding down **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 and . In the same way you can move the cursor left using 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 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.

Holding down 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 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  enters the Matrix Mode Debugger.

When switching on the MEGA65 or pressing the reset button on the side, while holding down  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,       and  to perform functions with a single press.

Hold  to access  through to  as shown on the front of each Function key.

Only Function keys  to  are available in C64 mode.

## HELP

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

## ALT

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

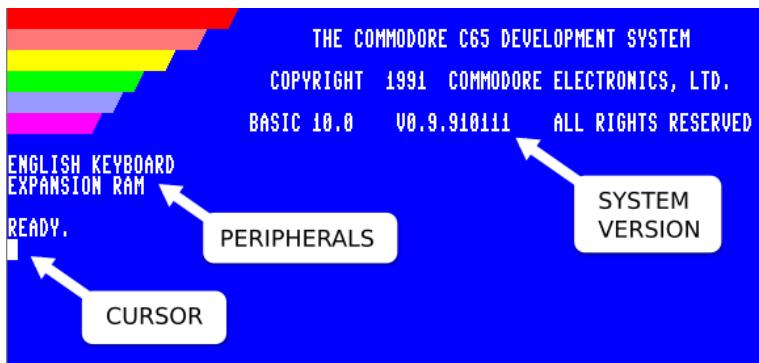
Holding  down when switching the MEGA65 on activates the Utility Menu. You can format the SD card or enter the MEGA65 Configuration Utility to select the default video mode and other settings, or test your keyboard.

## 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 de-packing process after a program is run. This can reduce the loading and de-packing 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.

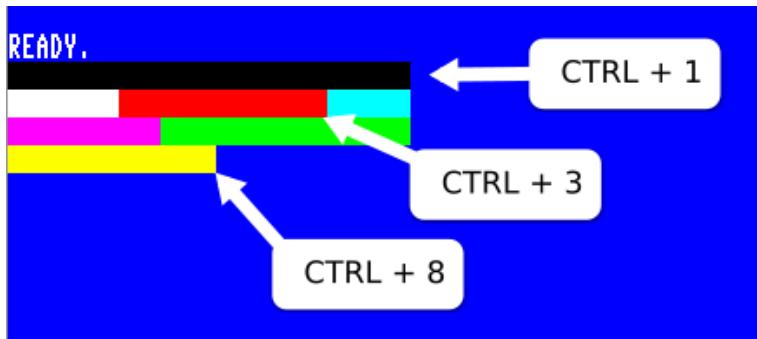
You can also produce reverse text or colour bars by holding down the **CTRL** key and pressing the **9** key or the **R** key. This enters reverse text mode.

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.

You can create fun pictures just using these colours and the letters. Here's an example of what a year four student drew:



What will you draw?

## 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 and releasing it, followed by 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

## Configuring your MEGA65

- Preparing for First Use: Formatting SD Cards
- Installing ROM and Other Support Files
- Configuring your MEGA65



# PREPARING FOR FIRST USE: FORMATTING SD CARDS

The MEGA65 has two SD card slots: A full-size SD card slot inside, next to the trap-door, and a microSD size slot on the rear. The current version of the MEGA65 firmware only supports the use of one SD card at a time. If you have cards in both slots, the MEGA65 will use the external microSD slot in preference. The exception to this, is that the MEGA65's FDISK/FORMAT utility can access both, allowing you to select which you wish to format or repair.

Depending on the model, your MEGA65 may or may not come with a pre-configured SD card. If it doesn't, or if you wish to use a different SD card, e.g., with a larger capacity, you must first format it for use in the MEGA65.

*This must be done in the MEGA65, not in a PC or other computer.*

*Only SDHC cards should be used. Older SD cards (typically with a capacity of <4GB) will not work. Newer SDXC cards with capacities greater than 32GB may or may not work. We would appreciate hearing your experience with such cards. It is unimportant what file-system is currently on the card, as the MEGA65 FDISK/FORMAT utility will completely reformat the card.*

There are several reasons for this: First, in order to fit the most features into the MEGA65's small operating system, it is particular about the FAT32 file system it uses. Second, only the MEGA65 FDISK/FORMAT utility can create a MEGA65 System Partition. The MEGA65 System Partition holds non-volatile configuration settings for your MEGA65, and also contains the freeze slots, that make it easy to switch which programme or game you are running on your MEGA65.

Fortunately, formatting an SD card on the MEGA65 is very easy.

First, power the MEGA65 on while holding down the **ALT** key. This will present the MEGA65 Utility Menu, which contains a selection of built-in utilities, similar to the following:

# MEGA65

MEGAS5 MEGROS HYPERVISOR V00.15

```
GIT: 138-HD.3EC382B+DIRTY.20200512.20
NO_SCROLL=FLASH, ALT=UTILS, CTRL=HOLD
GIT: 138-HD.3EC382B+DIRTY.20200512.20
SELECT UTILITY TO LAUNCH
1. CONFIGURE MEGAS5
2. SDCARD FDISK+FORMAT UTILITY
3. KEYBOARD TEST
```

The exact set of utilities depends on the model of your MEGA65 and the version of the MEGA65 factory core which it is running. However, all versions include both the MEGA65 FDISK/FORMAT utility, and the MEGA65 Configure utility. Most models also include a keyboard test utility, that you can use to test that your keyboard is functioning correctly. This is the same utility used in the factory when testing brand new keyboards.

Select the number that corresponds to the FDISK/FORMAT utility. This will typically be 2. The FDISK utility will start, and attempt to detect the size of all SD cards you have installed. If you have both an internal and external SD card installed, it will allow you to choose which one you wish to format. The internal SD card is bus 1, and the external card is bus 0. Note that the MEGA65 will always attempt to boot using an external microSD card, if one is installed.

For safety when formatting we *strongly* recommend that you remove any SD card or microSD card that you do not intend to format, so that you do not accidentally destroy any data. This is because formatting an SD card in the MEGA65 cannot be undone, and you all data currently on the SD card *will be lost*. If you have files or data on the SD card that you wish to retain, you should first back this up. The contents of the FAT32 partition can be easily backed up by inserting the SD card into another computer. The contents of the MEGA65 System Partition, including the contents of freeze slots requires the use of specialised software.

You should aim to backup valuable data from your MEGA65 on a regular basis, especially while the computer remains under development. While we take every care to avoid data corruption or other mishaps, we cannot guarantee that the MEGA65 is free of bugs in this regard.

If you have only an internal SD card, you might see a display similar to the following:

```
Detecting SD card(s) (can take a while)
SD Card 0 (External MicroSD slot):
Maximum readable sector is $00EEB7FE
7638 MiB SD CARD FOUND.
SD Card read speed = 1152 KB/sec

Current partition table:
0C : Start=0 /0 /0      or 00000000 / End=0 /0 /0      or 00AEAFFE
41 : Start=0 /0 /0      or 00EB7FE / End=0 /0 /0      or 00400000
00 : Start=0 /0 /0      or 00000000 / End=0 /0 /0      or 00000000
00 : Start=0 /0 /0      or 00000000 / End=0 /0 /0      or 00000000
```

```
SD Card 1 (internal SD slot):
No card detected on bus 1
```

```
Please select SD card to modify: 0/1
```

```
MEGA65 FDISK+FORMAT V00.08 : (C) COPYRIGHT 2017-2019 PAUL GARDNER-STEPHEN ETC.
```

Once you have selected the bus, the FDISK/FORMAT utility asks you to confirm that you wish to delete everything:

```
Please select SD card to modify: 0/1
Maximum readable sector is $00EEB7FE
7638 MiB SD CARD FOUND,
SD Card read speed = 1154 KB/sec

Current partition table:
0C : Start=0 /0 /0      or 00000000 / End=0 /0 /0      or 00AEAFFE
41 : Start=0 /0 /0      or 00AEB7FE / End=0 /0 /0      or 00400000
00 : Start=0 /0 /0      or 00000000 / End=0 /0 /0      or 00000000
00 : Start=0 /0 /0      or 00000000 / End=0 /0 /0      or 00000000

$00400000 Sectors available for MEGA65 System partition.
2046 Freeze and OS Service slots.
$00AEAFFE Sectors available for VFAT32 partition.

Format Card with new partition table and FAT32 file system?
5589 MiB VFAT32 Data Partition @ 00000000:
$0015CAD5 Clusters, 11158 Sectors/FAT, 568 Reserved Sectors.
2048 MiB MEGA65 System Partition @ 00AEB7FE:

Type DELETE EVERYTHING to continue:
Or type FIX MBR to re-write MBR

MEGA65 FDISK+FORMAT V00.08 : (C) COPYRIGHT 2017-2019 PAUL GARDNER-STEPHEN ETC.
```

To avoid accidentally loss of data, you must type “DELETE EVERYTHING” in capitals and press **RETURN**. Alternatively, turn the MEGA65 off and on to abort this process without causing damage to your data.

It is also possible to attempt to recover from a lost Master Boot Record (“Boot Sector”) by instead typing “FIX MBR,” should the need arise.

## INSTALLING ROM AND OTHER SUPPORT FILES

The MEGA65 FDISK/FORMAT utility will install a version of the open-source OpenROM project’s C64-compatible ROMs as part of the formatting process. However, you may have other ROMs that you wish to use on the MEGA65. The 911001 version of the C65 ROM in particular is known to work well with the MEGA65. You can copy as many of these as you wish onto the SD card. Make sure that they have the .ROM extension. The default ROM should be called MEGA65.ROM. These files should be 128KB in size, and use the same internal format as ROMs intended for the C65. This means that the C64-mode

KERNAL should be placed at offset \$E000, a C65-mode BASIC at \$A000, and a suitable character set at \$D000.

Other important files include FREEZER.M65 and AUDIOMIX.M65, which allow you to use the MEGA65's integrated freezer. You can download the full set of support files for the MEGA65 from:

<https://github.com/mega65/mega65-files>

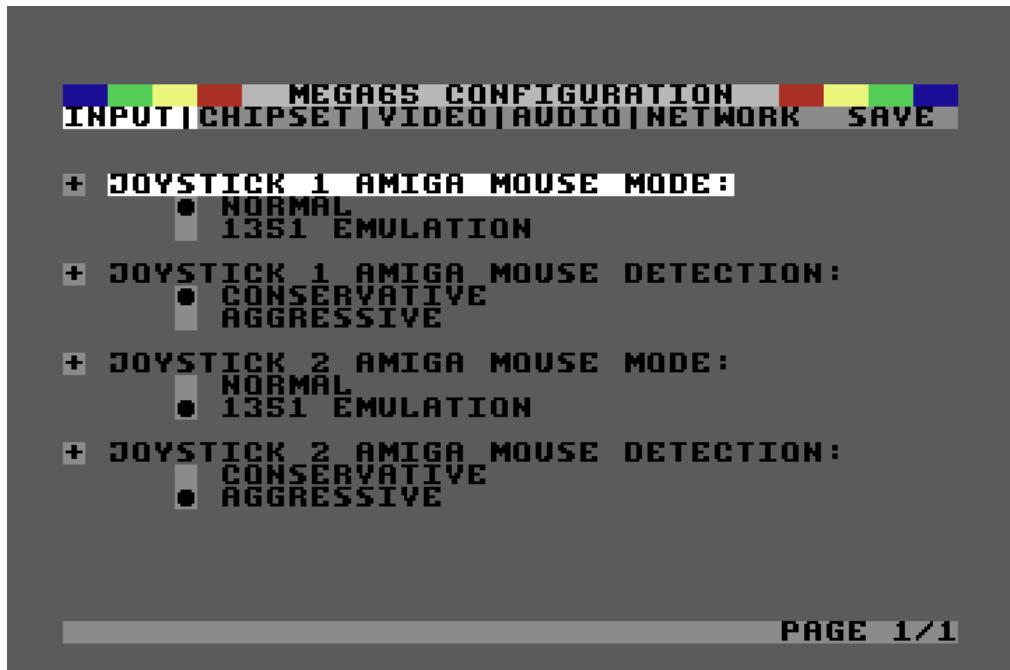
## CONFIGURING YOUR MEGA65

The configuration utility for the MEGA65 fills a similar purpose to the BIOS on a PC, and allows you to control certain default behaviours of your MEGA65. However, rather than storing the configuration data in a battery-backed RAM, it stores them on sector 1 of the SD card. This means that if you switch SD cards, you will change the configuration data that you are using.

To enter the configuration utility, turn the MEGA65 on while holding the **ALT** key. This will show the utility menu, similar to the following:



Now press the number corresponding to the Utility Menu. The MEGA65 Configuration Utility will launch, showing a display similar to the following:



If your MEGA65's System Partition has become corrupt, you may be prompted to press **F14** to correct this, i.e., hold **SHIFT** and tap the **F13** key, with a display like the following:



If you do, you will need to use **F7** to save the reset configuration, as otherwise the reset data will not be saved to the MEGA65 System Partition.

Once you have dismissed that display, or if your MEGA65 System Partition was not corrupted, you can begin exploring and adjusting various settings. The programme can be controlled using the keyboard, or optionally, an Amiga(tm) or C1351 mouse.

You can advance screens by pressing **F1**, or use **F2** to navigate in the opposite direction through the screens. You can also use the **←** and **→** keys to navigate between screens.

The **↑** and **↓** keys can be used to select an item.

Press **RETURN** or **SPACE** to toggle a setting, or to allow changing a text or numeric value. The black circle next to an option indicates that it is the selected setting.

When finished, you can press **F7** to receive the option to save the changes. This will give you four options:

**MEGA65 CONFIGURATION**

**INPUT|CHIPSET|VIDEO|AUDIO|NETWORK      SAVE**

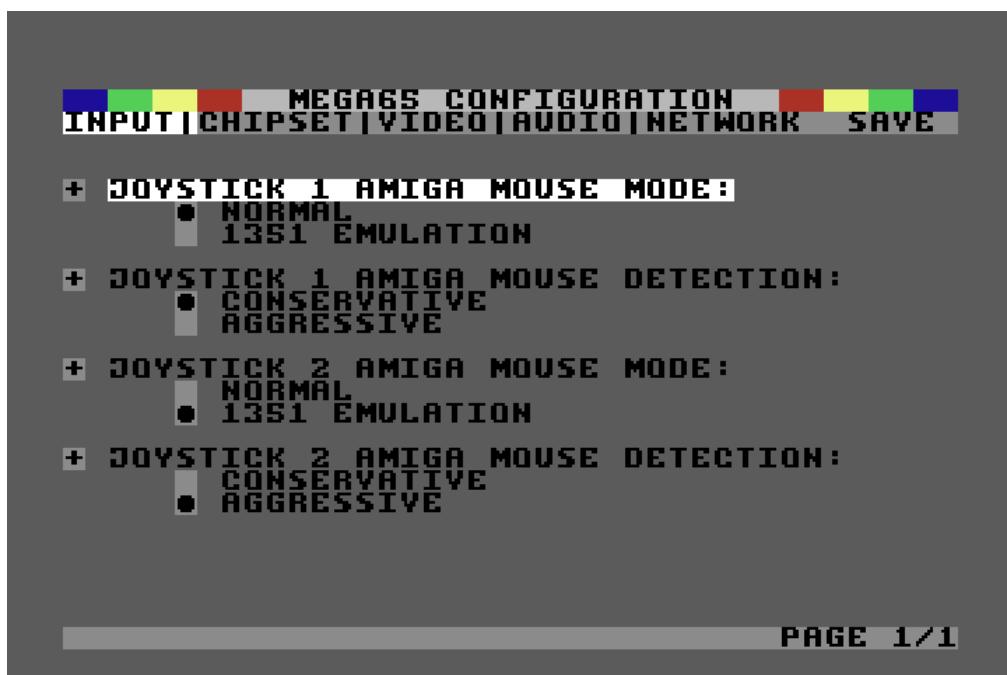
- + EXIT WITHOUT SAVING**
- + APPLY AND TEST SETTINGS NOW**
- + RESTORE FACTORY DEFAULTS**
- + SAVE AS DEFAULTS AND EXIT**

**ANY KEY FOR DATA ENTRY**

**PAGE 1/1**

- *Exit Without Saving* allows you to abandon any changes made in the MEGA65 Configure utility, without saving them.
- *Apply and Test Settings Now* makes the settings take immediate effect. This can be helpful for testing compatibility of your TV or monitor with PAL or NTSC video modes. If you still see your display after applying such a change, it is safe to save those settings.
- *Restore Factory Defaults* allows you to reset the MEGA65 configuration settings to the factory defaults. It also randomly selects a new MAC address for models that include an internal Ethernet adaptor. If you wish to commit these changes, you must still save them.
- *Save as Default and Exit* commits any changes you have made to the SD card storage, so that they will be used whenever your MEGA65 is turned on.

## Input Devices



- *Joystick 1 Amiga Mouse Mode* allows either **normal** operation, where software will see it as an Amiga mouse, or **1351 emulation** mode, where the MEGA65 translates the Amiga mouse's movements into 1351 compatible signals. This allows you to use an Amiga mouse with existing C64/C65 software that expect a 1351 mouse.
- *Joystick 1 Amiga Mouse Detection* can be set to conservative or aggressive. If you use an Amiga mouse, and it fails to move smoothly in all directions, you may set it to **aggressive**. Conversely, if you regularly use joysticks in the port, and have difficulties with the joystick input mis-behaving, you may select the **conservative** option.
- *Joystick 2 Amiga Mouse Mode* is the same as the first option, but for the second joystick port. This allows you to have different policies for each port.
- *Joystick 2 Amiga Mouse Detection* similarly provides the ability to separately control the Amiga mouse detection algorithm for the second joystick port.

## Chipset



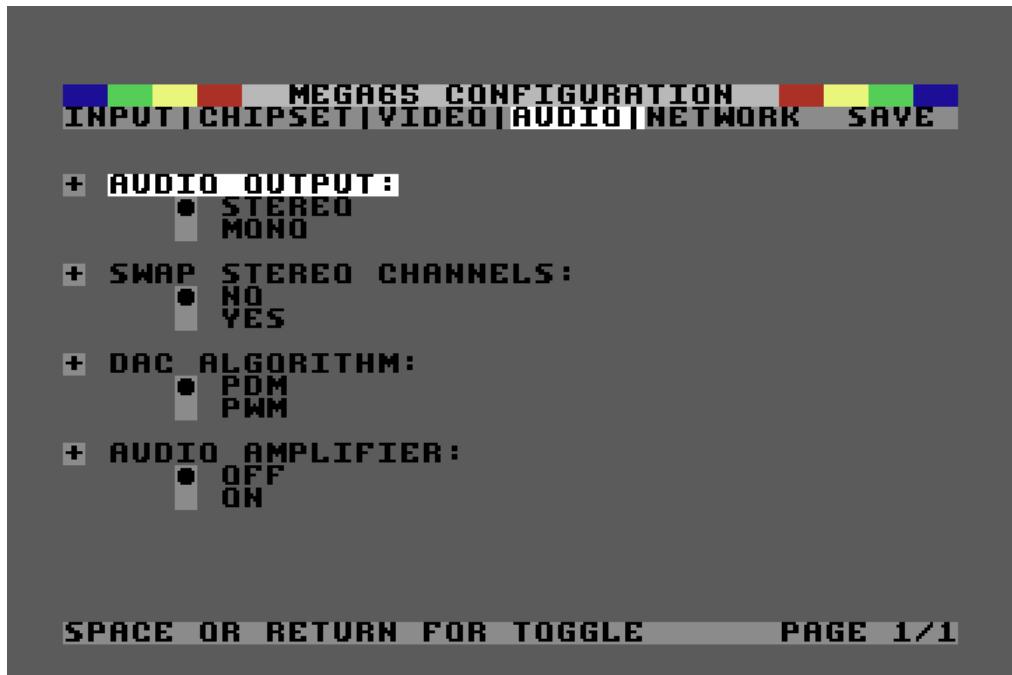
- The Real-Time Clock allows setting the MEGA65's Real-Time Clock for those models that include one. To set the clock or calendar, simply edit the field and press the **RETURN** key. The display does not change while viewing this page, but if you use the cursor left and right keys to select another page and return to this page, the values will update if a Real-Time Clock is fitted and functioning.
- *DMAagic Revision* allows selecting the default mode of operation for the C65 DMAagic DMA controller. This option is only required for ROMs not detected by the MEGA65's HYPPO Hypervisor. If you see screen corruption in BASIC, try toggling this option.
- *F011 Disk Controller* This option allows you to select whether the internal 3.5" floppy drive functions using real diskettes, or whether it simply makes noises to add atmosphere when using D81 disk images from the SD card. This merely sets the default option, and you can change this setting, or select a different disk image for use as either or both of the C65 3.5" DOS based drives.
- *Default Disk Image* allows you to choose the D81 disk image used with the internal drive, if the F011 Disk Controller option above is set to use an SD card disk image.

## Video



- *Video Mode* selects whether the MEGA65 starts in PAL or NTSC. The MEGA65 supports true 480p NTSC and 576p PAL double-scan modes, with exact 60Hz / 50Hz frame-rates. This setting sets the default value, and the system can be switched between PAL and NTSC via the Freeze Menu, or under software control by MEGA65-enabled programmes.

## Audio



- *Audio Output* selects whether the SIDs and digital audio channels are combined to provide a mono-aural signal, or whether the left and right tagged audio sources are separated to provide a stereo signal. Again, this setting can be varied from in the Audio Mixer of the Freeze Menu, or under the control of MEGA65-enabled software.
- *Swap Stereo Channels* allows switching the left and right sides of the stereo audio output. This is primarily useful for software that expects left and right SIDs to be at swapped addresses compared with the MEGA65.
- *DAC Algorithm* allows selecting between two different digital to analog conversion algorithms. Both are very good, but you may have a preference for one or the other.
- *Audio Amplifier* allows enabling or disabling the audio amplifier contained in some models of the MEGA65 for certain audio outputs, e.g., internal speaker or loud speaker.

## Network



- *MAC Address* allows you to set the default MAC address of your MEGA65. This can be changed at run-time by MEGA65-enabled software



# 5

## CHAPTER

# Cores and Flashing

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

- **Understanding The Core Booting Process**





# 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 very 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 **NO SCROLL** 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:

```
(0) MEGA65 FACTORY CORE  
(1) EMPTY SLOT  
(2) Pre-Series August 2019  
18  
(3) MEGA65 Unstable Branch  
20200303.19-UNSTAB-F39032F+dirt  
(4) EMPTY SLOT  
(5) HDMI Test Grid Display  
2000305.15-UNSTAB-SA6212F  
(6) EMPTY SLOT  
(7) EMPTY SLOT  
0-7 = Launch Core.  CTRL 1-7 = Edit Slot
```

To select a core and start it, use the cursor keys to highlight the desired core, and then press the **RETURN** key. If you select a flash slot that does not contain a valid core, it will highlight in red to indicate that it cannot be booted from:

**(0) MEGA65 FACTORY CORE**

**(1) EMPTY SLOT**

**(2) Pre-Series August 2019  
18**

**(3) MEGA65 Unstable Branch  
20200303.19-UNSTAB-F39D32F+dirt**

**(4) EMPTY SLOT**

**(5) HDMI Test Grid Display  
2000305.15-UNSTAB-SA6212F**

**(6) EMPTY SLOT**

**(7) EMPTY SLOT**

**0-7 = Launch Core. CTRL 1-7 = Edit Slot**

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 faulty, 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.

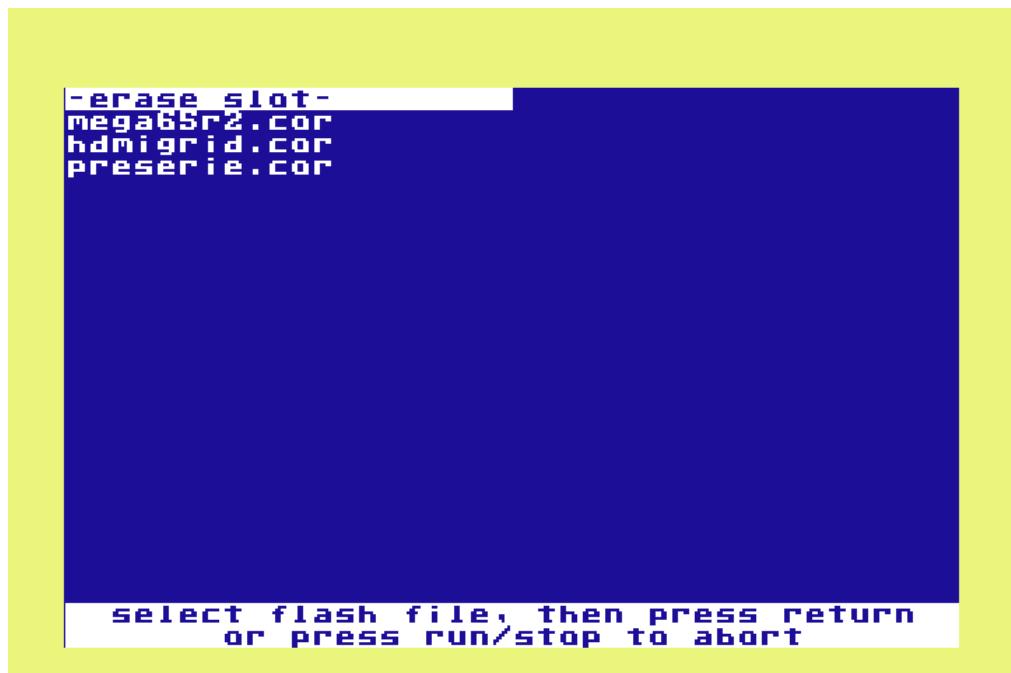
## INSTALLING AN UPGRADE CORE FOR THE MEGA65

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.

The flash menu will preferentially use the external microSD slot over the internal SD card: If you have both a microSD card and SD card inserted in your MEGA65, the flash menu will currently ignore the internal SD card. Simply copy the core file(s) from the internal SD card to the external microSD card, or temporarily remove the external microSD card from the rear of your MEGA65, so that the flash menu will be able to find the core files. Also note that the flash menu currently supports only DOS-style 8.3 character filenames. If your core files have a longer name, you will need to rename them when copying them onto your microSD or SD card.

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 **NO SCROLL** key down while turning the power on. When the flash and core menu appears, hold the **CTRL** key down and press the **1** key (or **CTRL** and a different number if you wish to replace the contents of a different flash slot). The MEGA65 will present you a list of core files that are on the SD card, similar to this:



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. You will see a progress bar while the MEGA65 erases the flash slot, similar to this:



```
Erasing flash slot...
sector at $00436000 erased.
```

The progress bar will then reset, and the MEGA65 will write the new core into the slot. This process can take up to 15 minutes, depending on the size of the core file. If you simply wish to erase a flash slot, you can select the “- erase slot -” option instead of a file name. This will perform only the erasure part of the process.

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. While inconvenient, if this happens, it won’t damage your MEGA65 or leave it in an unusable state: It will just fall back to using the factory supplied core. If this happens, enter the flash and core menu as described above, and follow the instructions again.

When the flashing process completes, you will see a message like this, indicating that the process is complete:

```
Erasing flash slot...
&Flash slot successfully written.
Press any key to return to menu.
```

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 CORES

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 O 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 adaptor, and then 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.

You have been warned!

## UNDERSTANDING THE CORE BOOTING PROCESS

This section summarises how the MEGA65 selects which core to start when it is powered on. The process is as shown in the following figure: When the MEGA65 is powered on, it always starts the bitstream stored in slot 0 of the flash. If that is the MEGA65 Factory Core, the MEGA65 HYPPO Hypervisor starts. If it is the first boot since power-on, HYPPO

starts the Flash Menu programme – but note that the Flash Menu in this mode may not show anything on the screen to indicate that it is running!

The Flash Menu checks if the system is booting from Flash Slot 0. If it is, it checks if the NO SCROLL key is being held. If it is, the Flash Menu programme shows its display, allowing the user to select or re-flash a core. If the NO SCROLL key is not being pressed, then the Flash Menu programme checks if Flash Slot 1 contains a valid core. If it does, then the Flash menu programme attempts to start that core. If it succeeds, then the system reconfigures to that core, after which the behaviour of the system is according to that core. If it fails, the keyboard will show the flashing red and blue “ambulance lights” to indicate that some first-aid is required. Note that in the ambulance light mode that the reset button has no effect: You must turn the computer off and on again.

If the user selected a different core in the Flash Menu, the process is similar, except that the ambulance lights will appear for only a limited time, as the FPGA will automatically search through the flash memory until it finds a valid core. If it gets to the end of the flash memory, it will start the MEGA65 Factory Core from slot 0 again.

# PART

## FIRST STEPS IN CODING



# CHAPTER

# 6

## 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:

<b>Number in the egg-cup</b>	<b>Action</b>
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.

# CHAPTER 7

## Getting Started in BASIC

- Your first BASIC programmes
- First steps with text and numbers
- Making simple decisions
- Random numbers and chance



It is possible to code on the MEGA65 in many languages, however most people start with BASIC. That makes sense, because BASIC stands for Beginner's All-purpose Symbolic Instruction Code: It was made for people like you to get started with in the world of coding!

A few short words before we dive in: BASIC is a programming language, and like spoken language it has conventions, grammar and vocabulary. Fortunately, it is much quicker and easier to learn than our complex human languages. But if you pay attention, you might notice some of these structures, and that can help you along your path in the world of coding.

If you haven't already read Chapter 3, it might be a good idea to do so. This will help you be able to more confidently interact with the MEGA65 computer.

It's also great to remember that if you really confuse the MEGA65, you can always get back to the READY. prompt by just pressing the reset button on the left-hand side of the keyboard, or if that doesn't help, then by turning it off and on again using the power switch on the left-hand side of the keyboard. You don't have to worry about shutting the computer down properly or any of that nonsense. The only thing to remember is that if you had any unsaved work, it will be lost when you turn the computer off and on again or press the reset button.

Finally, if you don't understand all of the descriptions and information with an example – don't worry! We have provided as much information as we can, so that it is there in case you have questions, encounter problems are just curious to discover more. Feel free to skip ahead to the examples and try things out, and then you can go back and re-read it when you are motivated to find something out, or help you work though a problem. And if you don't find the answer to your problem, send us a message! There are support forums for the MEGA65 at <https://mega65.net>, and you can report problems with this guide at:

<https://github.com/mega65/mega65-user-guide>

We hope you have as much fun learning to programme the MEGA65 as we have had making it!

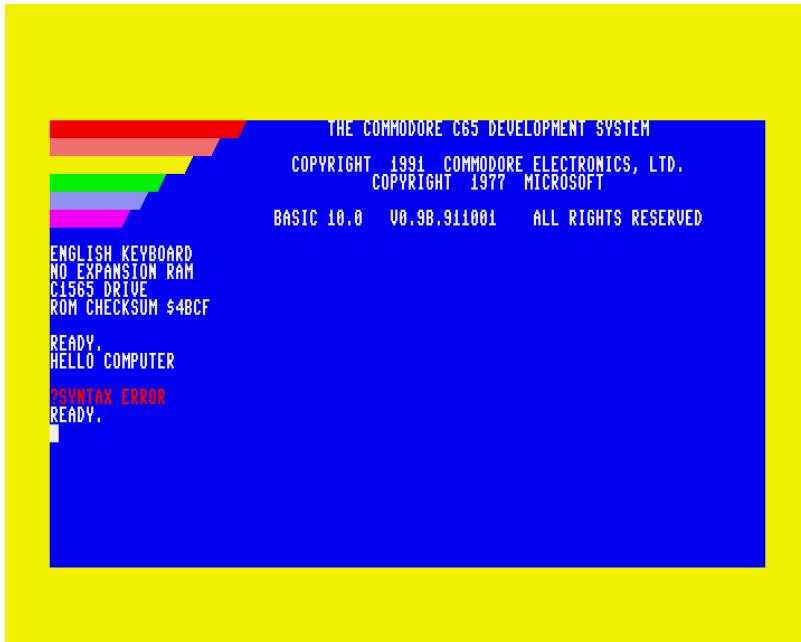
## YOUR FIRST BASIC PROGRAMMES

The MEGA65 was designed to be programmed! When you turn it on, it takes a couple of seconds to get its house in order, and then it quickly shows you a "READY." prompt and flashing block called the cursor. When the cursor is blinking, it tells you that the computer is waiting for input. The "READY." message tells you that the BASIC programming language is running and ready for you to start programming. You don't even need to load any programmes – you can just get started.

Try typing the following into the computer and see what happens:

HELLO COMPUTER

To do this, just type the letters as you see them above. The computer will already be in upper-case mode, so you don't need to hold the **SHIFT** or **CAPS LOCK** key down. When you have typed "HELLO COMPUTER" press the **RETURN** key. This tells the computer you want it to accept the line of input you have typed. When you do this, you should see a message something like the following:



If you saw a **SYNTAX ERROR** message something like that one, then congratulations: You have succeeded in communicating with the computer! Error messages sound much nastier than they are. The MEGA65 uses them, especially the syntax error to tell you when it is having trouble understanding what you have typed, or what you have put in a programme. They are nothing to be afraid of, and experienced programmers get them all the time.

In this case, the computer was confused because it doesn't understand the word "hello" or the word "computer". That is, it didn't know what you wanted it to do. In this regard, computers are quite stupid. They know only a few words, and aren't particularly imaginative about how they interpret them.

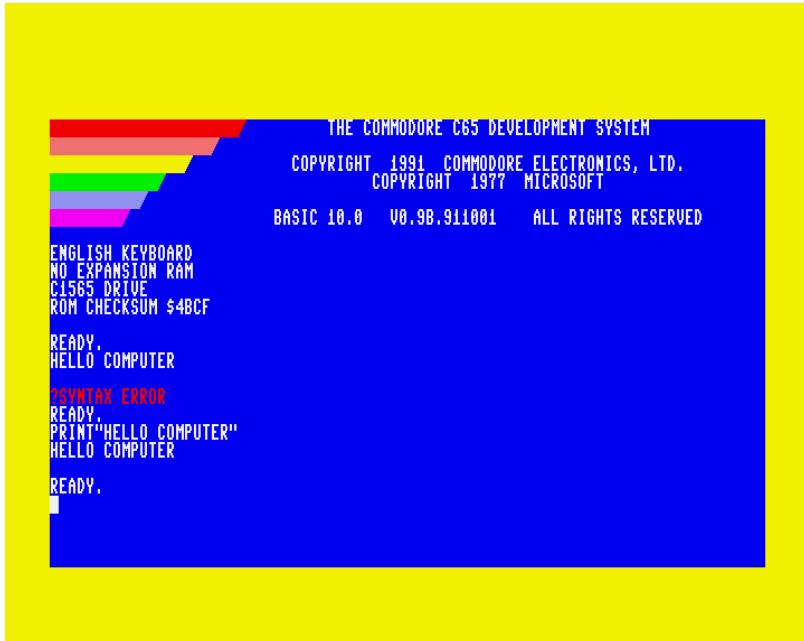
So let's try that again in a way that the computer will understand. Try typing the following in. You can just type it right away. It doesn't matter that the syntax error message can still be seen on the screen. The computer has already forgotten about that by the time it told you **READY.** again.

```
PRINT "HELLO COMPUTER"
```

Again, make sure you don't use shift or shift-lock while typing it in. The symbols around the words **HELLO COMPUTER** are double-quotes. If you are used to an Australian or American keyboard, you might discover that they double-quote key is in a rather different place to where you are used to: Double-quotes can be typed on the MEGA65 by holding down the **SHIFT** key, and then pressing 2. Don't forget to press the **RETURN** key when you are done, so that the computer knows you want it to do something with your input.

If you make a mistake while typing, you can use the **INST DEL** to rub out the mistake and fix it up. You can also use the cursor keys to move back and forth on the line while you edit the line you are typing, but there is a bit of a trick if you have already typed a double-quote: If you try to use the cursor keys, it will print a funny reversed symbol instead of moving the cursor. This is because the computer thinks you want to record moving the cursor in the text itself, which can be really useful and fun, and which you can read more about in Chapter 3. But for now, if you make a mistake just press the **RETURN** key and type the messed up line again.

Hopefully now you will see something like the following:

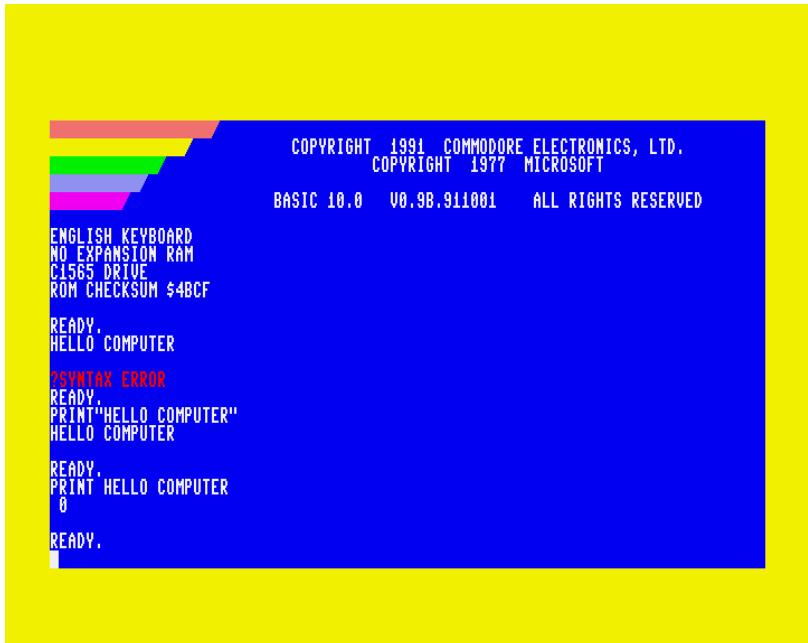


This time no new **SYNTAX ERROR** message should appear. But if some kind of error message has appeared, just try typing in the command again, after taking a close look to work out where the mistake might be.

Instead of an error, we should see **HELLO COMPUTER** repeated underneath the line you typed in. The reason this happened is that the computer does understand the word **PRINT**. It knows that whatever comes after the word **PRINT** should be printed to the screen. We had to put **HELLO COMPUTER** inside double-quotes to tell the computer that we want it to be printed literally.

If we hadn't put the double-quotes in, the computer would have thought that **HELLO COMPUTER** was the name of a stored piece of information. But because we haven't stored any piece of information in such a place, the computer will have zero there, so the computer will print the number zero. If the computer prints zero or some other number when you expected a message of some sort, this can be the reason.

You can try it, if you like, and you should see something like the following:



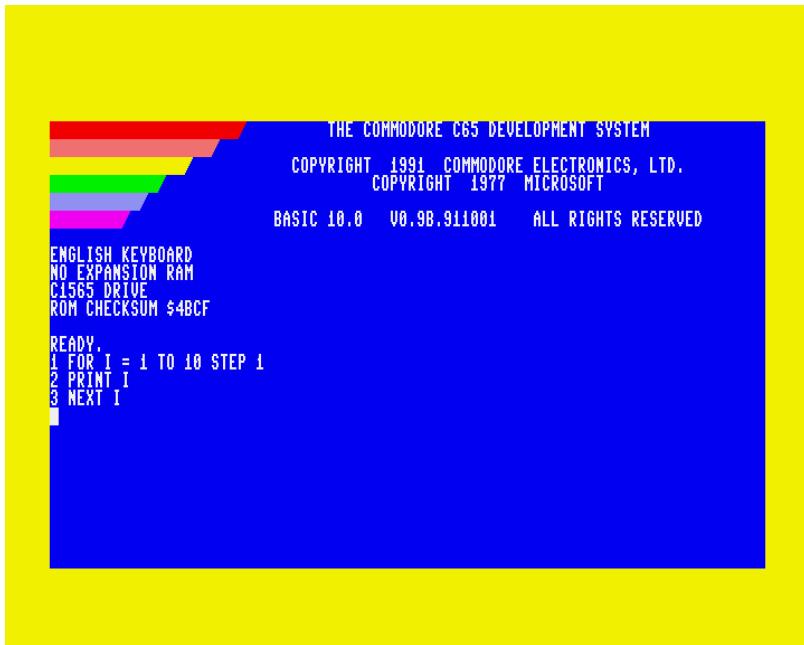
In the above examples we typed commands in directly, and the computer executed them immediately after you pressed the **RETURN** key. This is why typing commands in this way is often called *direct mode* or *immediate mode*.

But we can also tell the computer to remember a list of commands to execute one after the other. This is done using the rather unimaginatively named *non-direct mode*. To use non-direct mode, we just put a number between 0 and 63999 at the start of the command. The computer will then remember that command. Unlike when we executed a direct-mode command, the computer doesn't print **READY.** again. Instead the cursor just reappears on the next line, ready for us to type in more commands.

Let's try that out with a simple little programme. Type in the following three lines of input:

```
1 FOR I = 1 TO 10 STEP 1  
2 PRINT I  
3 NEXT I
```

When you have done this, the screen should show something like this:



If it doesn't work, you can try again. Don't forget, if you feel that the computer is getting all muddled up, you can just press the reset button or flip the power switch off and on on the left side of the computer to reboot it. This only takes a couple of seconds, and doesn't hurt the MEGA65 in anyway.

We have told the computer to remember three commands, that is, **FOR I = 1 TO 10**, **STEP 1**, **PRINT I** and **NEXT I**. We have also told the computer which order we would like to run them in: The computer will start with the command with the lowest number, and execute each command that has the next higher number in turn, until it reaches the end of the list. So it's a bit like a reminder list for the computer. This is what we call a programme, a bit like the programme at a concert or the theatre, it tells us what is coming up, and in what order. So let's tell the computer to execute this programme.

But first, let's try to guess what will happen. Let's start with the middle command, **PRINT I**. We've seen the **PRINT** command, and we know it tells the computer to print things to the screen. The thing it will try to print is **I**. Just like before, because there are no double-quotes around the **I**, it will try to print a piece of stored information. The piece of information it will try to print will be the piece associated with the thing **I**.

When we give a piece of information like this a name, we call it a *variable*. They are called variables because they can vary. That is, we can replace the piece of information

associated with the variable called **I** with another piece of information. The old piece will be forgotten as a result. So if we gave a command like **LET I = 3**, this would replace whatever was stored in the variable called **I** with the number 3.

Back to our programme, we now know that the 2<sup>nd</sup> command will try to print the piece of information stored in the variable **I**. So lets look at the first command: **FOR I = 1 TO 10 STEP 1**. Although we haven't seen the **FOR** command before, we can take a bit of a guess at how it works. It looks like it is going to put something into the variable **I**. That something seems to have something to do with the range of number 1 through 10, and a step or interval of 1. What do you think it will do?

If you guessed that it will put the values 1, 2, 3, 4, 5, 6, 7, 8, 9 and then 10 into the variable **I**, then you can give yourself a pat on the back, because that's exactly what it does. It also helps us to understand the 3<sup>rd</sup> command, **NEXT I**: That command tells the computer to put the next value into the variable **I**. And here is a little bit of magic: When the computer does that, it goes back up the list of commands, and continues again from the command after the **FOR** command.

So lets pull that together: When the computer executes the first command, it discovers that it has to put 10 different values into the variable **I**. It starts by putting the first value in there, which in this case will be the number 1. The computer then continues to the second command, which tells the computer to print the piece of information that is currently stored in the variable called **I**. That will be the number 1, since that was the last thing the computer was told to put there. Then the computer proceeds to the third command, which tells it that it is time to put the next value into the variable **I**. So the computer will throw away the number 1 that is currently in the variable **I**, and put the number 2 in there, since that is the next number in the list. It will then continue from the 2<sup>nd</sup> command, which will cause the computer to print out the contents of the variable **I** again. Except that this time **I** has had the number 2 stored in it most recently, so the computer will print the number 2. This process will repeat, until the computer has printed all ten values that the **FOR** command indicated it to do.

To see this in action, we need to tell the computer to execute the programme of commands we typed in. We do this by using the **RUN** command. Because we want it to run the programme immediately, we should use immediate mode (remember, this is another name for direct mode). So just type in the word **RUN** and press the **RETURN** key. You should then see a display that looks something like the following:



You might notice a couple of things here:

First, the computer has told us it is **READY**. again as soon as it finished running the programme. This just makes it easier for us to know when we can start giving commands to the computer again.

Second, when the computer got to the bottom of the screen it automatically scrolled the display up to make space. This is quite normal. What is important to remember, is that the computer forgets everything that scrolls off the top. The only exception is if you have told the computer to remember a command by putting a number in front of it. So our programme is quite safe for now. We can see that this is the case by typing the **RUN** command a couple more times: The programme listing will have scrolled off the top of the screen, but we can still RUN the programme, because the computer has remembered it. Give it a try! Did it work?

If you wish to see the programme of remembered commands, you can use the **LIST** command. This command causes the computer to display the remembered programme of commands to the screen, like in the display here. If you would like to replace any of the commands in the programme, you can type a new line that has the same number as the one you wish to change.

```
9  
10  
READY.  
RUN  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
READY.  
LIST  
1 FOR I = 1 TO 10 STEP 1  
2 PRINT I  
3 NEXT I  
READY.
```

For example, to print the results all on one line, we could modify the second line of the programme to **PRINT I;** by typing the following line of input and pressing the **RETURN** key:

```
2 PRINT I;
```

You can make sure that the change has been remembered by running the **LIST** command again, as we can see here. You can then use the **RUN** command to run the modified programme, like this:

```
7
8
9
10

READY.
LIST

1 FOR I = 1 TO 10 STEP 1
2 PRINT I
3 NEXT I

READY.
2 PRINT I;
LIST

1 FOR I = 1 TO 10 STEP 1
2 PRINT I;
3 NEXT I

READY.
RUN
1 2 3 4 5 6 7 8 9 10
READY.
```

It is quite easy to modify your programmes in this way. As you become more comfortable with the process, there are two additional helpful tricks:

First, you can give the **LIST** command the number of a command, or line as they are referred to, and it will display only that line of the programme. Alternatively, you can give a range separated by a minus sign to display only a section of the programme, e.g., **LIST 1 - 2** to list the first two lines of our programme.

Second, you can use the cursor keys to move the cursor to a line which has already been remembered and is displayed on the screen. If you modify what you see on the screen, and then press the **RETURN** key while the cursor is on that line, the BASIC interpreter will read in the modified line and replace the old version of it. It is important to note that if you modify multiple lines of the programme at the same time, you must press the **RETURN** key on each line that has been modified. It is good practice to check that the programme has been correctly modified. Use the **LIST** command again to achieve this.

## **Exercises to try**

### **1. Can you make it count to a higher or lower number?**

At the moment it counts from 1 to 10. Can you change it to count to 20 instead? Or to count from 3 to 17? Or how about from 14.5 to 21.5? What do you think you would need to reverse the order in which it counts?

*Clue:* You will need to modify the **FOR** command.

### **2. Can you change the counting step?**

At the moment it counts by ones, i.e., each number is one more than the last. Can you change it to count by twos instead? Or by halves, so that it counts 1, 1.5, 2, 2.5, 3, ...?

*Clue:* You will need to modify the **STEP** clause of the **FOR** command.

### **3. Can you make it print out one of the times tables?**

At the moment it prints the answers to the 1 times tables, because it counts by ones. Can you make it count by threes, and show the three times tables?

*Clue:* You will need to modify the **FOR** command.

### **4. Can you make it print out the times tables from $1 \times 1$ to $10 \times 10$ ?**

*Clue:* You might like to use ; on the end of **PRINT** command, so that you can have more than one entry per line on the screen.

*Clue:* The **PRINT** command without any argument will just advance to the start of the next line.

*Clue:* You might need to have multiple **FOR** loops, one inside the other.

## **FIRST STEPS WITH TEXT AND NUMBERS**

In the last section we started to use both numbers and text. Text on computers is made by stringing individual letters and other symbols together. For this reason they are called *strings*. We also call the individual letters and symbols *characters*. The name character comes from the printing industry where each of the symbols that can be printed on a page. For computers, it has much the same meaning, and the set of characters that a computer can display is rather unimaginatively called a *character set*.

When the MEGA65 expects some form of input, it is typically looking for one of four things:

1. a keyword like **PRINT** or **STEP**, which are words that have a special meaning to the computer;
2. a variable name like **I** or **A\$** that it will then use to either store or retrieve a piece of information;

3. a number like **42** or **-30.3137**; or

4. a string like "**HELLO COMPUTER**" or "**23 KILOMETRES**".

Sometimes you have a choice of which sort of thing you can provide, while other times you have less choice. What sort of thing the computer will accept depends on what you are doing at the time. For example, in the previous section we discovered that when the computer tells us that it is **READY**, that we can give it a keyword or a number. Do you think that the computer will accept all four kinds of thing when it says **READY**.? We already know that keywords and numbers and keywords can be entered, but what about variable names or strings? Let's try typing in a variable name, say **N**, and pressing the **RETURN** key, and see what happens. And then lets try with a string, say "**"THIS IS A STRING"**".



You should get a syntax error each time, telling you that the computer doesn't understand the input you have given it. Let's start with when you typed the variable: If you just tell the computer the name of a stored piece of information, it doesn't have the foggiest idea what you are wanting it to do. It's the same when you give it a piece of information, like a string, without telling the computer what to do with it.

But as we discovered in the last section, we can tell the computer that we want to see the piece of information that is stored in a variable using the **PRINT** command. So we could instead type in **PRINT N**, and the computer would know what to do, and will print the piece of information stored in the variable called **N**.

In fact, using the **PRINT** command is so common, that programmers got annoyed having to type in the **PRINT** command all the time, that they made a short cut: If you type a question mark character, i.e., a ?, the computer knows that you mean **PRINT**. So for example if you type ? **N**, it will do the same as typing **PRINT N**. Of course, you have to press the **RETURN** key after each command to tell the computer you want it to process what you typed. From here on, we will assume that you can remember to do that, without being reminded.

The ? shortcut also works if you are telling the computer to remember a command as part of a programme. So if you type **I ? N**, and then **LIST**, you will see **I PRINT N**, as we can see in the following screen-shot:

```
READY.  
N  
SYNTAX ERROR  
READY.  
"THIS IS A STRING"  
SYNTAX ERROR  
READY.  
PRINT N  
0  
READY.  
? N  
0  
READY.  
I ? N  
LIST  
I PRINT N  
READY.
```

Like we saw in the last section, the variable **N** has not had a value stored in it, so when the computer looks for what is there, it finds nothing. Because **N** is a *numeric variable*, when there is nothing there, this means zero. If it was a *string variable*, then it would have found literally nothing. We can try that, but first we have to explain how we tell the computer we are talking about a string variable. We do that by putting a dollar sign character, i.e., a \$, on the end of the variable name. So if we put a \$ on the end of the variable name **N**, it will refer to a string variable called **N\$**.

We can experiment with these variables by using the hopefully now familiar **PRINT** command (or the **?**  shortcut) to see what is in the variables. But we need a convenient way to put values into them. Fortunately we aren't the first people to want to put values into variables, and so the **LET** exists. The **LET** command is used to put a value into a variable. For example, we can tell the computer:

```
LET N = 5.3
```

This tells the computer to put the value 5.3 into the variable **N**. We can then use the **PRINT** command to check that it worked. Similarly, we can put a value into the variable **N\$** with something like:

```
LET N$ = "THE KING OF THE POTATO PEOPLE"
```

If we try those, we will see something like the following:

```
? N
0
READY.
1 ? N
LIST
1 PRINT N
READY.
LET N = 5.3
READY.
? N
5.3
READY.
LET N$ = "THE KING OF THE POTATO PEOPLE"
READY.
? N$
THE KING OF THE POTATO PEOPLE
READY.
```

We mentioned just before that **N** is a numeric variable and that **N\$** is a string variable. This means that we can only put numbers into **N** and strings into **N\$**. If we try to put the wrong kind of information into a variable, the computer will tell us that we have mis-matched the kind of information with the place we are trying to put it by giving us a **TYPE MISMATCH ERROR** like this:

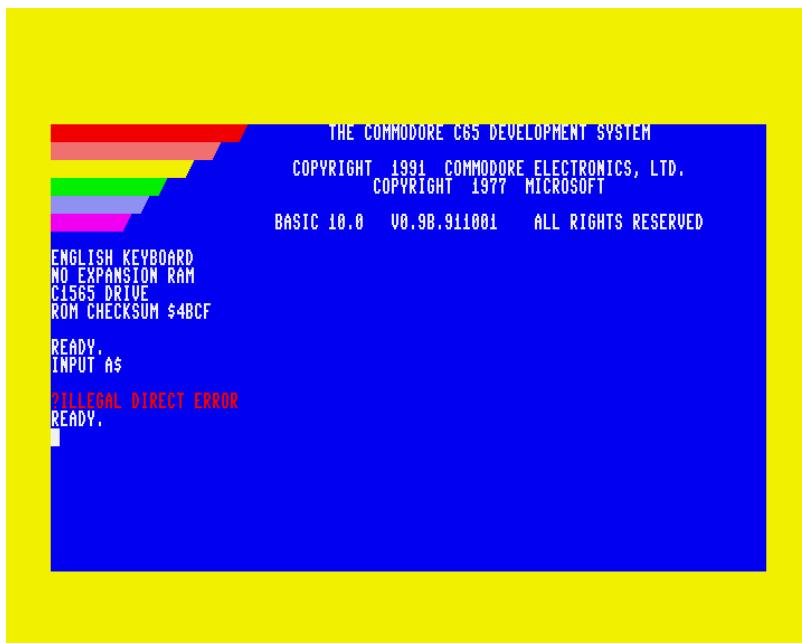
```
READY.  
LET N = 5.3  
  
READY.  
? N  
5.3  
  
READY.  
LET N$ = "THE KING OF THE POTATO PEOPLE"  
  
READY.  
? N$  
THE KING OF THE POTATO PEOPLE  
  
READY.  
LET N = "MR FLIBBLE"  
  
?TYPE MISMATCH ERROR  
READY.  
LET N$ = 42  
  
?TYPE MISMATCH ERROR  
READY.
```

This leads us to a rather important point: **N** and **N\$** are separate variables, even though they have similar names. This applies to all possible variable names: If the variable name has a **\$** character on the end, it means it is a string variable quite separate from the similarly named numeric variable. To use a bit of jargon, this means that each *type* of variable has their own separate *name spaces*.

(There are also four other variable name spaces that we haven't talked about yet: integer variables, identified by having a **%** character at the end of their name, e.g., **N%**, and arrays of numeric, string or integer variables. But don't worry about those for now. We'll talk about those a bit later on.)

So far we have only given values to variables in direct mode, or by using constructions like **FOR** loops. But we haven't seen how we can get information from the user when a programme is running. One way that we can do this, is with the **INPUT** command.

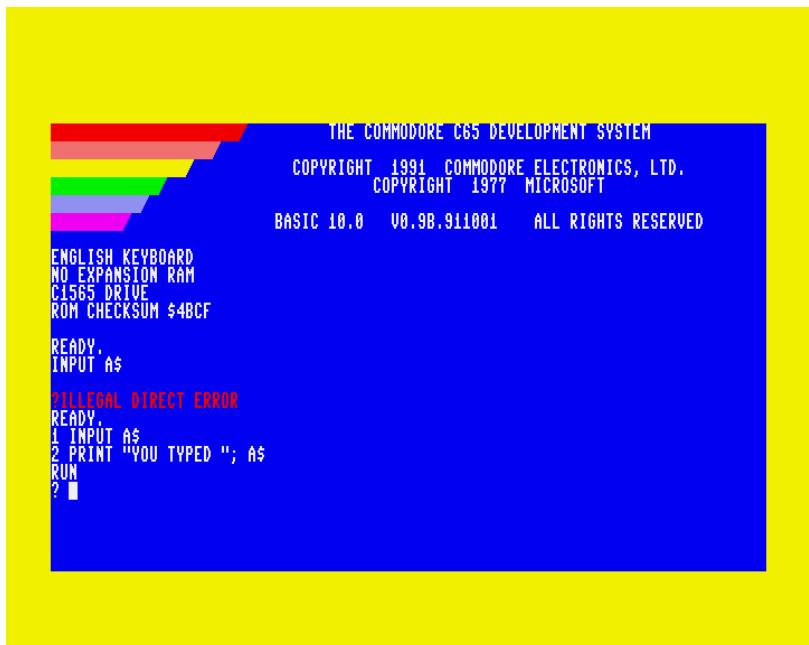
**INPUT** is quite easy to use: We just have to say which variable we would like the input to go into. For example, to tell the computer to ask for the user to provide something to put into the variable **A\$**, we could use something like **INPUT A\$**. The only trick with the **INPUT** command is that it cannot be used in direct mode. If you try it, the computer will tell you **ILLEGAL DIRECT ERROR**. Try it, and you should see something like the following



This means that the **INPUT** command can only be used as part of a programme. So we can instead do something like the following:

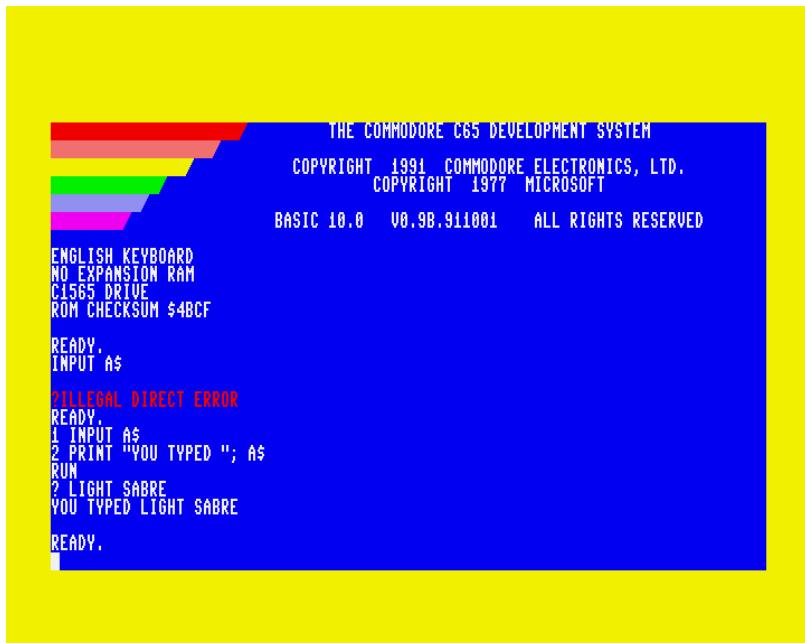
```
1 INPUT A$  
2 PRINT "YOU TYPED "; A$  
RUN
```

What do you think that this will do? The first line will ask the computer for something to put into the variable **A\$**, and the second line will print the string "**"YOU TYPED"**", followed by what the **INPUT** command read from the user. Let's try it out:



Did you expect that to happen? What is this question mark doing there? The **?** here is the computer's way of telling you that a programme is waiting for some input from you. This means that the computer uses the same symbol, **?**, to mean two different things: If you type it as part of a programme or in direct mode, then it is a short-cut for the **PRINT** command. That's when you type it. But if the computer shows it to you, it has this other meaning, that the computer is waiting for you to type something in. There is also a third way that the computer uses the **?** character. Have you noticed what it is? It is to indicate the start of an error message. For example, a Syntax Error is indicated by **?SYNTAX ERROR**. When a character or something has different meanings in different situations or contexts, we say that it is *context dependent*.

But returning to our example, if we now type something in, and press the **RETURN** key to tell the computer that you are done, the programme will continue, like this:



Of course, we didn't really know what to type in, because the programme didn't give any hints to the user as to what the programmer wanted them to do. So we should try to provide some instructions. For example, if we wanted the user to type their name, we could print a message asking them to type their name, like this:

```
1 PRINT "WHAT IS YOUR NAME"  
2 INPUT A$  
3 PRINT "HELLO "; A$
```

Now if we run this programme, the user will get a clue as to what we expect them to do, and the whole experience will make a lot more sense for them:

```
ROM CHECKSUM $4BCF
READY.
INPUT A$  
ILLEGAL DIRECT ERROR
READY.
1 INPUT A$  
2 PRINT "YOU TYPED "; A$  
RUN  
? LIGHT SABRE  
YOU TYPED LIGHT SABRE

READY.
1 PRINT "WHAT IS YOUR NAME"  
2 INPUT A$  
3 PRINT "HELLO "; A$  
RUN  
WHAT IS YOUR NAME  
? LISTER  
HELLO LISTER

READY.
```

When we run the programme, we first see the **WHAT IS YOUR NAME** message from line 1. The computer doesn't print the double-quote symbols, because they only told the computer that the piece of information between them is a string. The string itself is only the part in between.

After this we see the **?** character again and the blinking cursor telling us that the computer is waiting for some input from us. The rest of the programmed is *blocked* from continuing until it we type the piece of information. Once we type the piece of input, the computer stores it into the variable **A\$**, and can continue. Thus when it reaches line 3 of the programme, it has everything it needs, and prints out both the **HELLO** message, as well as the information stored in the variable called **A\$**.

Notice that the word **LISTER** doesn't appear anywhere in the programme. It exists only in the variable. This ability to process information that is not part of a programme is one of the things that makes computer programmes so powerful and able to be used for so many purposes. All we have to do is to change the input, and we can get different output.

For example, with our programme we run it again and again, and give it different input each time, and the programme will adapt its output to what we type. Pretty nifty, right? Let's have the rest of the crew try it out:

```
WHAT IS YOUR NAME
? LISTER
HELLO LISTER

READY.
RUN
WHAT IS YOUR NAME
? HOLLY
HELLO HOLLY

READY.
RUN
WHAT IS YOUR NAME
? KRYPTON
HELLO KRYPTON

READY.
RUN
WHAT IS YOUR NAME
? RIMMER, BSC
?EXTRA IGNORED
HELLO RIMMER

READY.
```

We can see that each time the programme prints out the message customised with the input that you typed in...Until we get to **RIMMER, BSC**. As always, Mr. Rimmer is causing trouble. In this case, he couldn't resist putting his Bronze Swimming Certificate qualification on the end of his name.

We see that the computer has given us a kind of error message, **?EXTRA IGNORED**. The error is not written in red, and doesn't have the word **ERROR** on the end. This means that it is a warning, rather than an error. Because it is only a warning, the programme continues. But something has happened: The computer has ignored Mr. Rimmer's **BSC**, that is, it has ignored the extra input. This is because the **INPUT** command doesn't really read a whole line of input. Rather, it reads *one piece of information*. The **INPUT** command thinks that a piece of information ends at the end of a line of input, or when it encounters a comma (,) or colon (:) character.

If you want to include one of those symbols, you need to surround the whole piece of information in double-quotes. So, if Mr. Rimmer had read this guide instead of obsessing over the Space Core Directives, he would have known to type "**"RIMMER, BSC"**" (complete with the double-quotes), to have the programme run correctly. It is important that the quotes go around the whole piece of information, as otherwise the computer will think that the first quote marks the start of a new piece of information. We can see the difference it makes below:

```
LIST
1 PRINT "WHAT IS YOUR NAME"
2 INPUT A$
3 PRINT "HELLO "; A$

READY.
RUN
WHAT IS YOUR NAME
? "RIMMER, BSC"
HELLO RIMMER, BSC

READY.
RUN
WHAT IS YOUR NAME
? "RIMMER" " BSC
?EXTRA IGNORED
HELLO RIMMER"

READY.
```

While this can all be a bit annoying at times, it has a purpose: The **INPUT** command can be used to read more than one piece of information. We do this by putting more than one variable after the **INPUT** command, each separated by a comma. The **INPUT** command will then expect multiple pieces of information. For example, we could ask for someone's name and age, with a programme like this:

```
1 PRINT "WHAT IS YOUR NAME AND AGE"
2 INPUT A$, A
3 PRINT "HELLO "; A$
4 PRINT "YOU ARE"; A; " YEARS OLD."
```

If we run this programme, we can provide the two pieces of information on the one line when the computer presents us with the **?** prompt, for example **LISTER, 3000000**. Note the comma that separates the two pieces of information, **LISTER** and **3000000**. It's also worth noticing that we haven't put any thousands separators into the number 3,000,000. If we did, the computer would think we meant three separate pieces of information, **3**, **000** and **000**, which is not what we meant. So let's see what it looks like when we give **LISTER, 3000000** as input to the programme:

The image shows a computer terminal window with a yellow background and a blue foreground. The text inside the window is as follows:

```
READY.  
RUN  
WHAT IS YOUR NAME  
? RIMMER" " BSC  
?EXTRA IGNORED  
HELLO RIMMER"  
  
READY.  
LIST  
  
1 PRINT "WHAT IS YOUR NAME AND AGE"  
2 INPUT A$,A  
3 PRINT "HELLO "; A$  
4 PRINT "YOU ARE"; A; " YEARS OLD."  
READY.  
  
RUN  
WHAT IS YOUR NAME AND AGE  
? LISTER, 3000000  
HELLO LISTER  
YOU ARE 3000000 YEARS OLD.  
  
READY.
```

In this case, the **INPUT** command reads the two pieces of information, and places the first into the variable **A\$**, and the second into the variable **A**. When the programme reaches line 3 it prints **HELLO** followed by the first piece of information. Then when it gets to line 4, it prints the string **YOU ARE**, followed by the contents of the variable **A**, which is the number 3,000,000, and finally the string **YEARS OLD**.

It's also possible to just give one piece of information at a time. In that case, the **INPUT** command will ask for the second piece of information with a double question-mark prompt, i.e., **??**. Once it has the second piece of information. (If we had more than two variables on the **INPUT** command, it will still present the same **??** prompt, rather than printing more and more question-marks.)

So if we try this with our programme, we can see this **?**  and **??** prompts, and how the first piece of information ends up in **A\$** because it is the first variable in the **INPUT** command. The second piece of information ends up in **A** because **A** is the second variable after the **INPUT** command. Here's how it looks if we give this input to our programme:

```
READY.  
LIST  
1 PRINT "WHAT IS YOUR NAME AND AGE"  
2 INPUT A$,A  
3 PRINT "HELLO "; A$  
4 PRINT "YOU ARE"; A; " YEARS OLD."  
READY.  
  
RUN  
WHAT IS YOUR NAME AND AGE  
2 LISTER,3000000  
HELLO LISTER  
YOU ARE 3000000 YEARS OLD.  
  
READY.  
RUN  
WHAT IS YOUR NAME AND AGE  
2 LISTER  
?? 3000000  
HELLO LISTER  
YOU ARE 3000000 YEARS OLD.  
  
READY.
```

Until now we have been asking the user to input information by using a **PRINT** command to display the message, and then an **INPUT** command to tell the computer which variables we would like to have some information input into. But, like with the **PRINT** command, this is something that happens often enough, that there is a shortcut for it. It also has the advantage that it looks nicer when running, and makes the programme a little shorter. The short cut is to put the message to show after the **INPUT** command, but before the first variable.

We can change our programme to use this approach. First, we can change line 3 to include the prompt after the **INPUT** command. We can do this one of two ways: First, we could just type in a new line 3. The computer will automatically replace the old line 3 with the new one.

But, as we have mentioned a few times now, programmers are lazy beasts, and so there is a short-cut: If you can see the line on the screen that you want to change, you can use the cursor keys to navigate to that line, edit it on the screen, and then press the **RETURN** key to tell the computer to accept the new version of the line.

Either way, you can check that the changes succeeded by typing the **LIST** command on any line of the screen that is blank. This will show the revised version of the programme. For example:

```
LIST
1 PRINT "WHAT IS YOUR NAME AND AGE"
2 INPUT A$,B
3 PRINT "HELLO "; A$
4 PRINT "YOU ARE"; B; " YEARS OLD."
READY
2 INPUT "WHAT IS YOUR NAME AND AGE",A$,A
LIST
1 PRINT "WHAT IS YOUR NAME AND AGE"
2 INPUT "WHAT IS YOUR NAME AND AGE",A$,A
3 PRINT "HELLO "; A$
4 PRINT "YOU ARE"; B; " YEARS OLD."
READY.
```

We still have a little problem, though: Line 1 will print the message **WHAT IS YOUR NAME AND AGE**, and then Line 2 will print it again! We only want the message to appear once. Thus we would like to change line 1 so that it doesn't do this any more. Because there is no other command on line 1 that we want to keep, that line can just become empty. So we can type in something like this:

We can confirm that the contents of the line have been deleted by running the **LIST** command again, like this:

```
1 PRINT "WHAT IS YOUR NAME AND AGE"
2 INPUT A$,B
3 PRINT "HELLO "; A$
4 PRINT "YOU ARE"; B; " YEARS OLD."
READY.
2 INPUT "WHAT IS YOUR NAME AND AGE",A$,B
LIST
1
2 INPUT "WHAT IS YOUR NAME AND AGE",A$,B
3 PRINT "HELLO "; A$
4 PRINT "YOU ARE"; B; " YEARS OLD."
READY.
```

Did you notice something interesting? When we told the computer to make line 1 of the programme empty, it deleted it completely! That's because the computer thinks that an empty line is of no use. It also makes sure that your programmes don't get all cluttered up with empty lines if you make lots of changes to your programmes.

With that out the way, let's run our programme and see what happens. As usual, just type in the **RUN** command and hit the **RETURN** key. You should see something like this:

```
1 PRINT "WHAT IS YOUR NAME AND AGE"
2 INPUT A$,B
3 PRINT "HELLO ",A$,
4 PRINT "YOU ARE"; A; " YEARS OLD."
READY.
2 INPUT "WHAT IS YOUR NAME AND AGE",A$,A
LIST
1 PRINT "WHAT IS YOUR NAME AND AGE"
2 INPUT "WHAT IS YOUR NAME AND AGE",A$,A
3 PRINT "HELLO ",A$,
4 PRINT "YOU ARE"; A; " YEARS OLD."
READY.
1
LIST
2 INPUT "WHAT IS YOUR NAME AND AGE",A$,A
3 PRINT "HELLO ",A$,
4 PRINT "YOU ARE"; A; " YEARS OLD."
READY.
RUN
WHAT IS YOUR NAME AND AGE|
```

We can see our prompt of **WHAT IS YOUR NAME AND AGE** there, but now the cursor is appearing without any **?** character. This is because we put a comma (,) after the message in the **INPUT** command. To get the question mark, we have to instead put a semi-colon (;) after the message, like this:

```
INPUT "WHAT IS YOU NAME AND AGE"; A$, A
```

Now if we run the programme, we should see what we are looking for:

```
LIST
2 INPUT "WHAT IS YOUR NAME AND AGE";A$,A
3 PRINT "HELLO "; A$;
4 PRINT "YOU ARE"; A; " YEARS OLD."
READY.
RUN
WHAT IS YOUR NAME AND AGE? LISTER,3000000
HELLO LISTER
YOU ARE 3000000 YEARS OLD.
READY.
```

### **Exercises to try**

- 1. Can you make the programme ask someone for their name, and then for their favourite colour?**

At the moment it asks for their name and age. Can you change the programme so that it reports on their favourite colour instead of their age?

*Clue:* What type of information is age? Is it numeric or a string? Is it the same type of information as the name of a colour?

- 2. Can you write a programme that asks someone for their name, prints the hello message, and then asks for their age and prints out that response?**

At the moment, the programme expects both pieces of information at the same time. This means the programme can't print a message about the first message until after it has both pieces of information. Change the programme so that you can have an interaction like the following instead:

```
WHAT IS YOUR NAME? DEEP THOUGHT  
HELLO DEEP THOUGHT  
WHAT IS THE ANSWER? 42  
YOU SAID THE ANSWER IS 42
```

*Clue:* You will need more lines in your programme, so that you can have more than one **INPUT** and **PRINT** command.

### 3. Can you write a programme that asks several questions, and then prints out the list of answers given?

Think of several questions you would like to be able to ask someone, and then write a programme that asks them, and remembers the answers and prints them out with an appropriate message. For example, running your programme could look like this:

```
WHAT IS YOUR NAME? FRODO  
HOW OLD ARE YOU? 33  
WHAT IS YOUR FAVOURITE FOOD? EVERYTHING!  
THANK YOU FOR ANSWERING.  
YOUR NAME IS FRODO  
YOU ARE 33 YEARS OLD  
YOU FAVOURITE FOOD IS EVERYTHING!
```

*Clue:* You will need more lines in your programme, to have the various **INPUT** and **PRINT** commands.

*Clue:* You will need to think carefully about which variable names you will use.

## MAKING SIMPLE DECISIONS

In the previous section we have learnt how to input text and numeric data, and how to display it. However, the programmes have just followed the lines of instruction in order, without any way to decide what to do, based on what has been input.

In this section we will see how we can take simple decisions using the **IF** and **THEN** commands. The **IF** command checks if something is true or false, and if it is true, causes the computer to execute the command that comes after the **THEN** command.

The way the computer decides whether something is true or false is that it operates on the supplied information using one of several symbols. These symbols are thus called *operators*. Also, because they compare two things, they depend on the relationship of the things. For this reason they are called *relational operators*. They include the following:

- Equals ( $=$ ). For example,  $3 = 3$  would be true, while  $3 = 2$  would be false.
- Less than ( $<$ ). For example,  $1 < 3$  would be true, while both  $3 < 3$  and  $1 < 3$  would be false.
- Greater than ( $>$ ). For example,  $3 > 1$  would be true, while both  $3 > 3$  and  $1 > 3$  would be false.

As it is common to want to consider when something might be equal or greater than, or equal or less than, there are short cuts for this. Similarly, if you wish to test if something is not equal to something else, there is a relational operator for this, too:

- Unequal, which we normally say as *not equal* ( $\neq$ ). This is different to the mathematical symbol for not equal,  $\neq$ , because the MEGA65's character set does not include a character that looks like that. So the programmers who created BASIC for the MEGA65 used the greater than and less than signs together to mean either less than or greater than, that is, not equal to. For example,  $1 \neq 3$  would be true, while  $3 \neq 3$  would be false.
- Less than or equal to ( $\leq$ ). For example,  $1 < 3$  and  $3 \leq 3$  would be true, while both  $4 < 3$  would be false.
- Greater than or equal to ( $\geq$ ). For example,  $3 \geq 1$  and  $3 \geq 3$  would be true, while both  $1 \geq 3$  would be false.

A good trick if you have trouble remembering which way the ( $<$ ) and ( $\geq$ ) signs go, the side with more ends of lines is the one that needs to have more. For example, the ( $<$ ) symbol has one point on the left, but two ends of lines on the right hand side. So for something to be true with ( $<$ ), the number on the left side needs to be less than the number on the right side. This trick even works for the equals sign, ( $=$ ), because it has the same number of ends on both sides, so you can remember that the numbers on both sides need to be equal. It also works when you have two symbols together, like ( $\geq$ ), it is true if the condition is true for any of the symbols in it. So in this case the ( $\geq$ ) symbol has more ends on the left than the right, so if the number on the left is bigger than the number on the right, it will be true. But also because the ( $=$ ) symbol has two ends on each side, it will be true if the two numbers are the same.

Using these relational operators, we can write a line that will do something, but only if something is true or false. Let's try this out, with a few examples:

```
IF -2 < 0 THEN PRINT "-2 IS A NEGATIVE NUMBER"
IF 2 < 0 THEN PRINT "2 IS A NEGATIVE NUMBER"
IF 0 < -2 THEN PRINT "-2 IS A POSITIVE NUMBER"
IF 0 < 2 THEN PRINT "2 IS A POSITIVE NUMBER"
```

These commands work fine in direct mode, so you can just type them directly into the computer to see what they will do. This can be handy for testing whether you have the logic correct when planning an **IF - THEN** command. If you type in those commands, you should see something like the following:



We can see that only the **PRINT** commands that followed an **IF** command that has a true value were executed. The rest were silently ignored by the computer. But we can of course include these into a programme. So lets make a little programme that will ask for two numbers, and say whether they are equal, or if one is greater or less than the other. Before you have a look at the programme, have a think about how you might do it, and see if you can figure it out. The clue I will give you, is that the **IF** command also accepts the name of a variables, not just numbers. So you can do something like **IF A > B THEN PRINT "SOMETHING"**. The programme will be on the next page, to stop you peeking before you have a think about it!

Did you have a go? There are lots of different ways it could be done, but here is what I came up with:

```
1 INPUT "WHAT IS THE FIRST NUMBER"; A  
2 INPUT "WHAT IS THE SECOND NUMBER"; B  
3 IF A = B THEN PRINT "THE NUMBERS ARE EQUAL"  
4 IF A > B THEN PRINT "THE FIRST NUMBER IS BIGGER"  
5 IF B > A THEN PRINT "THE SECOND NUMBER IS BIGGER"
```

We can then run the programme as often as we like, and the computer can tell us which of the two numbers we give it is biggest, or if they are equal:

```
1 INPUT "WHAT IS THE FIRST NUMBER"; A  
2 INPUT "WHAT IS THE SECOND NUMBER"; B  
3 IF A = B THEN PRINT "THE NUMBERS ARE EQUAL"  
4 IF A > B THEN PRINT "THE FIRST NUMBER IS BIGGER"  
5 IF B > A THEN PRINT "THE SECOND NUMBER IS BIGGER"  
  
RUN  
WHAT IS THE FIRST NUMBER? 2  
WHAT IS THE SECOND NUMBER? 2  
THE NUMBERS ARE EQUAL  
  
READY.  
RUN  
WHAT IS THE FIRST NUMBER? 3  
WHAT IS THE SECOND NUMBER? 4  
THE SECOND NUMBER IS BIGGER  
  
READY.  
RUN  
WHAT IS THE FIRST NUMBER? 10  
WHAT IS THE SECOND NUMBER? 2  
THE FIRST NUMBER IS BIGGER  
  
READY.
```

Notice how in this programme, we didn't use fixed numbers in the **IF** command, but instead gave variable names instead. This is one of the very powerful things in computer programming, together with being able to make decision based on data. By being able to refer to data by name, regardless of its current value or how it got there, lets the programmer create very flexible programmes.

Let's think about a bit of a more interesting example: a "guess the number" game. For this, we need to have a number that someone has to guess, and then we need to accept guesses, and indicate whether the guess was correct or not. If the guess is incorrect, we should tell the user if the correct number is higher or lower.

We have already learned most the ingredients to make such a program: We can use **LET** to set a variable to the secret number, **INPUT** to prompt the user for their guess, and then **IF**, **THEN** and **PRINT** to tell the user whether their guess was correct or not. So let's make something. Again, if you like, stop and think and experiment for a few minutes to see if you can make such a programme yourself.

Here is how I have done it. But don't worry if you have done it in a quite different way: There are often many ways to write a programme to perform a particular task.

```
1 SN=23
2 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"
3 INPUT "WHAT IS YOUR GUESS"; G
4 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"
5 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"
6 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"
```

The first line puts our secret number into the variable **SN**. The second line prints a message telling the user what they are supposed to do. The third line asks the user for their guess, and puts it into the variable **G**. The fourth, fifth and sixth lines then check whether the guess is correct or not, and if not, which message it should print. This is done by using the **IF** command and an appropriate relative operator to make each decision. This works well, to a point. For example:

```
LIST
1 SN=23
2 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"
3 INPUT "WHAT IS YOUR GUESS"; G
4 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"
5 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"
6 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"

READY.
RUN
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 10
MY NUMBER IS BIGGER

READY.
RUN
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 23
CONGRATULATIONS! YOU GUessed MY NUMBER!

READY.
```

We can see that it prints the message, and it asks for a guess, and responds appropriately. But if we want to guess again, we have to use the **RUN** command again for each extra guess. That's a bit poor from the user's perspective. However that is unlikely to be a problem for long, because the user can see the secret number in the listing on the screen!

So we would like to fix these problems. Let's start with hiding the listing. We previously mentioned that when the screen scrolls, anything that was at the top of the screen disappears. So we could just make sure the screen scrolls enough, that any listing that was visible is no longer visible. We could do this using **PRINT** and a **FOR** loop. The screen is 25 lines, so we could do something like:

```
FOR I = 1 to 25
PRINT
NEXT I
```

But there are better ways. If you hold down the **SHIFT** key, and then press the **CLR HOME** key, it clears the screen. This is much simpler and more convenient. But how can we do something like that in our programme? It turns out to be very simple: You can type it while entering a string! This is because the keyboard works differently based on whether you are in *quote mode*.

Quote mode is just a fancy way of describing what happens when you type a double-quote character into the computer: Until you type another double-quote or press the **RETURN**. You might remember we mentioned the problem of funny symbols coming up when using the cursor keys. I didn't want to distract you at the time, but that is a symptom of being in quote mode: In quote mode many special keys show a symbol that represents them, rather than taking their normal action. For example, if you press the cursor left key while in quote mode, a **↔** symbol appears. If you press the cursor right key, a **↔**, up **↑**, down **↓** and the **CLR HOME** a **⌫**, and if you are holding down **SHIFT** and press **CLR HOME** a **⌫**.

So let's use this to make the second line clear the screen when it prints the **GUESS THE NUMBER BETWEEN 1 AND 100** message. The first time you try it is a bit confusing, but once you get the hang of it, it is quite easy. What we want in the end is a line that looks like this:

```
2 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"
```

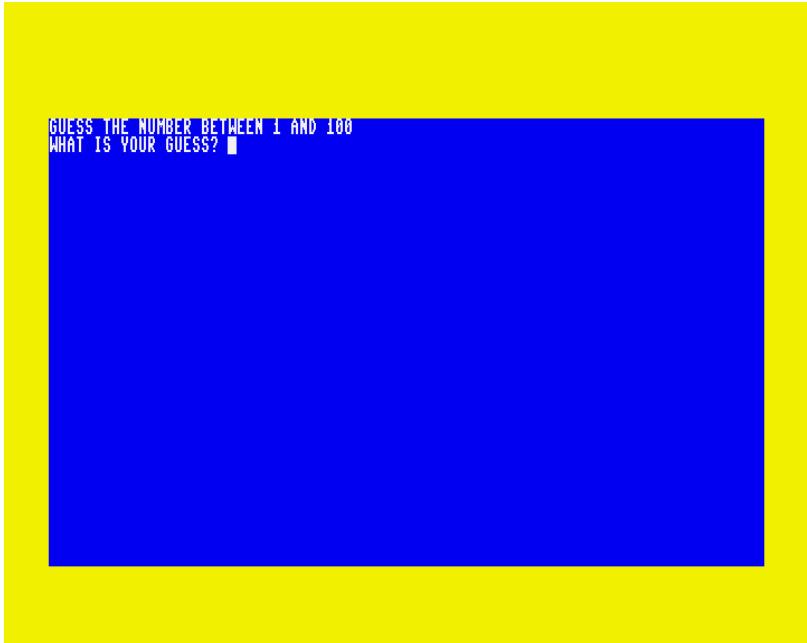
To do this, start by typing **2 PRINT** ". Then hold the **SHIFT** key down, and tap the **CLR HOME** key. Your line should now look like **2 PRINT"⌫"**. If so, you have succeeded! You can now finish typing the line as normal. When you have done that, you can use the **LIST** command as usual, to make sure that you have successfully modified the programme. You should see your modified line with the **⌫** symbol in it.

```
LIST
1 SN=23
2 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"
3 INPUT "WHAT IS YOUR GUESS";G
4 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"
5 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"
6 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"

READY.
2 PRINT"\"GUESS THE NUMBER BETWEEN 1 AND 100"
LIST
1 SN=23
2 PRINT"\"GUESS THE NUMBER BETWEEN 1 AND 100"
3 INPUT "WHAT IS YOUR GUESS";G
4 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"
5 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"
6 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"

READY.
```

If you now run the programme by typing in **RUN** and pressing the **RETURN** key as usual, the 2<sup>nd</sup> line tells the computer to clear the screen before printing the rest of the message, like this:



```
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? █
```

This hides the listing from the user, so that they can't immediately see what our secret number is. We can type our guess in, just like before, but just like before, after one guess, it returns to the **READY.** prompt. We really would like people to be able to make more than one guess, without needing to know that they need to run the programme again.

There are a few ways we could do this. We already saw the **FOR - NEXT** pattern. With that, we could make the programme give the user a certain number of guesses. If we followed the **NEXT** command with another programme line, we could even tell the user when they have taken too many guesses. So lets have a look at our programme and see how we might do that. Here is our current listing again:

```
1 SN=23
2 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"
3 INPUT"WHAT IS YOUR GUESS?"; G
4 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"
5 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"
6 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUESSED MY NUMBER!"
```

If we want the user to have multiple guesses, we need to have lines 2 through 6 run multiple times. This makes our life a bit tricky, because it means we need to insert a line between line 1 and 2. But unless you are a mathemagician, there are no whole numbers between 1 and 2, and the MEGA65 doesn't understand line numbers like 1.5.

Fortunately, the MEGA65 has the **RENUMBER** command. This command can be typed only in direct mode. When executed, it changes the line numbers in the programme, while keeping them in the same order. The new numbers are normally multiples of 10, so that you have lots of spare numbers in between to add extra lines. For example, if we use it on our programme, it will renumber the lines to 10, 20, ..., 60. We can see that this has happened by using the **LIST** command:

```
READY.  
LIST  
1 SN=23  
2 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"  
3 INPUT "WHAT IS YOUR GUESS";G  
4 IF G>SN THEN PRINT "MY NUMBER IS BIGGER"  
5 IF G<SN THEN PRINT "MY NUMBER IS SMALLER"  
6 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"  
  
READY.  
RENUMBER  
  
READY.  
LIST  
10 SN=23  
20 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"  
30 INPUT "WHAT IS YOUR GUESS";G  
40 IF G>SN THEN PRINT "MY NUMBER IS BIGGER"  
50 IF G<SN THEN PRINT "MY NUMBER IS SMALLER"  
60 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"  
  
READY.
```

Now our life is much easier: We can choose any number that is between 10 and 20 to put our **FOR** command into. It's a common choice to use the middle number, so that if you think of other things you want to add in later, you have the space to do it. So let's add a **FOR** command to give the user 10 chances to guess the number. We can use any variable name we like for this, except for **G** and **SN**, because we are using those. It would be very confusing if we mixed those up! So lets add a line like this:

```
15 FOR I = 1 TO 10 STEP 1
```

Now we need a matching **NEXT I** after line 60. Let's keep the nice pattern of adding 10 to work out the next line number, and put it as line 70:

```
70 NEXT I
```

We can type those lines in, and then use **LIST** command to make sure the result is correct:

```
LIST
10 SN=23
20 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"
30 INPUT "WHAT IS YOUR GUESS",G
40 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"
50 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"
60 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"

READY.
15 FOR I = 1 TO 10 STEP 1
70 NEXT I
LIST

10 SN=23
15 FOR I = 1 TO 10 STEP 1
20 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"
30 INPUT "WHAT IS YOUR GUESS",G
40 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"
50 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"
60 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"
70 NEXT I

READY.
```

That's looking pretty good. But there are a couple of little problems still. Can you work out what they might be? What will happen now after the user makes a guess? What will happen if they run out of guesses?

If you worked out that making a guess that the screen will be immediately cleared, you can give yourself a pat on the back! The user will hardly have time to see the message. Worse, if they guess the number correctly, they won't know, and the programme will keep going. We'd really like the programme to stop or end, once the user makes a correct guess.

We can do this using either the **STOP** or **END** commands. These two commands are quite similar. The main difference is that if you **STOP** a programme, the computer tells you where it has stopped, and you have the chance to continue the programme using the **CONT** command. The **END** command, on the other hand, tells the computer that the programme has reached its end, and it should go back to being **READY**. The **END** command makes

more sense for our programme, because after the user has guessed the number, there isn't any reason to continue.

Now we need a way to be able tell the computer to do two different things when the user makes a correct guess. We could just add an extra **IF** command after line 60 which prints the congratulations message, e.g., **65 IF G=SN THEN END**.

But we can be a bit more elegant than that: There is a way to have multiple commands on a single line. If you remember back to when we were learning about the **INPUT** command, you might remember that there were two different characters that separate pieces of information: , and :. The second one, :, is called a colon, and can also be used to separate BASIC commands on a single line. So if we want to change line 60 to **PRINT** the message of congratulations and then **END** the programme, we can just add : **END** to the end of the line. The line should look like this:

```
60 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!": END
```

That solves that problem. But it would also be nice to not clear the screen after every guess, so that the user can see what their last guess was, and whether it was bigger or smaller than the number. To do this, we can remove the clear-screen code from line 20, and add a new print command to a lower line number, so that it clears the screen once at the start of the programme, before the user gets to start guessing.

For example, we could put it in line 5, so that it happens as the absolute first action of the programme. As we mentioned earlier, the line numbers themselves aren't important: All that is important is to remember that the computer starts at the lowest line number, and runs the lines in order. Anyway, let's make those changes to our programme:

```
20 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"  
5 PRINT ""
```

If you type those lines in, and **LIST** the programme again, you should see something like the following:

```
40 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"  
50 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"  
60 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"  
70 NEXT I  
  
READY.  
  
20 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"  
5 PRINT "W"  
LIST  
  
5 PRINT "W"  
10 SN=23  
15 FOR I = 1 TO 10 STEP 1  
20 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"  
30 INPUT "WHAT IS YOUR GUESS"; G  
40 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"  
50 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"  
60 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUessed MY NUMBER!"  
70 NEXT I  
  
READY.
```

We can now **RUN** the programme, and see whether it worked. Let's try it!



The screen still clears, which is good. Can you notice one little difference already, though? There is a blank line above the first message. This is because our **PRINT** command in line 5 goes to the next row on the screen after it has printed the clear-screen character. We can fix this by putting a ; (semi-colon) character at the end of the **PRINT** command. This tells the **PRINT** command that it shouldn't go to the start of the next row on the screen when it has done everything. So if we change line 5 to **5 PRINT "█";** this will make the empty space at the top the screen disappear.

But back to our programme, we can now make guesses, and the programme will tell us whether each guess is more or less than the correct number. And after 10 guesses, it stops asking for guesses, and goes back to the **READY.** prompt, like this:

```
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 60
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 50
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 40
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 30
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 20
MY NUMBER IS BIGGER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 28
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 27
MY NUMBER IS SMALLER
READY.
```

It would be nice to tell the user if they have run out of guesses. We need to add this message after the **NEXT** command. We should also be nice and tell them what the secret number was, instead of leaving them wondering. So let's add the line to the end of our programme as line 80:

```
80 PRINT "SORRY! YOU RAN OUT OF GUESSES. MY NUMBER WAS"; SN
```

Now if the user doesn't guess the number, they will get a useful message, like this:

```
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 97
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 96
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 95
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 94
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 93
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 92
MY NUMBER IS SMALLER
GUESS THE NUMBER BETWEEN 1 AND 100
WHAT IS YOUR GUESS? 91
MY NUMBER IS SMALLER
SORRY! YOU RAN OUT OF GUESSES. MY NUMBER WAS 23
```

READY.

## Exercises to try

### 1. Can you make the programme ask at the start for the secret number?

At the moment the programme sets the secret number to 23 every time. To make the game more interesting it would be great to ask the first user for the secret number, and then start the rest of the game, so that someone else can try to guess the number.

*Clue:* You will need change the line that sets the **SN** variable so that it can be read from the first user. You might find the **INPUT** statement useful.

### 2. Can you make the programme ask for the user's name and give personalised responses?

At the moment, the programme displays very simple messages. It would be nice to ask the user their name, and then use their name to produce personalised messages, like **SORRY DAVE, BUT THAT NUMBER IS TOO SMALL.**

*Clue:* You will need to add a line early in the programme to ask the user their name.

*Clue:* You might like to review how we used the **PRINT** command, including with ; to print more than one thing on a line.

### **3. Can you improve the appearance of the messages with colours and better spacing?**

We haven't really made the programme particularly pretty. It would be great to use colours.

*Clue: You might like to add more **PRINT** commands to improve the spacing and layout of the messages.*

*Clue: You might like to use either the colour codes in the messages you **PRINT***

*Clue: You might also like to use the **FOREGROUND**, **BACKGROUND** and **BORDER** commands to set the colour of the text, screen background and border.*

### **4. Can you make the programme say if a guess is “warmer” or “colder” than the previous guess?**

At the moment the programme just tells you if the guess is higher or lower than the secret number. It would be great if it could tell you if a guess is getting closer or further away with each guess: When they get closer, it should tell the user that they are getting “warmer”, and “colder” when they get further away.

This is quite a bit more involved than the previous exercises, and requires you to work out some new things.

*Clue: You will need to remember the previous guess in a different variable, and then compare it with the last one: Is it nearer or further away. You might need to have **IF** commands that have another **IF** after the first one, or to learn how to use the **AND** operator.*

## RANDOM NUMBERS AND CHANCE

We'll come back to the Guess The Number game shortly, but let's take a detour first. Through a maze. Let's hope we can get back out before the end of the lesson! Let's look at a simple way to make a maze. This programme has been known for a long time. It works by choosing at random whether to display a **█** or a **▀** symbol. These symbols are obtained by holding down the **SHIFT** key and tapping either the N or M keys. You can see the symbols on the front of those keys. While they are shown on the keys with a box around them, the box does not appear, only the diagonal line. It turns out that printing either of these two characters at random draws a decent looking maze.

Let's give it a try. To be able to do this, we need a way to generate randomness. The MEGA65 has the **RND(1)** function to do this. This function works like a variable, but each time you try to use it, it gives a different result. Let's see how that works. Type in the following:

```
PRINT RND(1)
```

Each time you type this, it will give a different answer, as you can see here:

The screenshot shows a terminal window for the MEGA65 BASIC interpreter. The title bar reads "BASIC 10.0 V0.98.911001 ALL RIGHTS RESERVED". The screen displays the following text:

```
ENGLISH KEYBOARD
NO EXPANSION RAM
C1565 DRIVE
ROM CHECKSUM $4BCF

READY.
PRINT RND(1)
1.07870447E-03

READY.
PRINT RND(1)
.793262171

READY.
PRINT RND(1)
.44889513

READY.
PRINT RND(1)
.697215893

READY.
```

We can see that this gives us several different results: **1.07870447E-03**, **.793262171**, **.44889513**, **.697215893**. Each of these is a number between 0 and 1, even the first one. The first one is written in *scientific notation*. The **E-03** means that the value is  $1.07870447 \times 10^{-3} = 0.000107870447$ . That is, the **E-03** means to move the decimal place three places to the left. If there is a **+** after **E**, then it means to move the decimal place to the right. For example, **1.23456E+3** represents the number 1234.56.

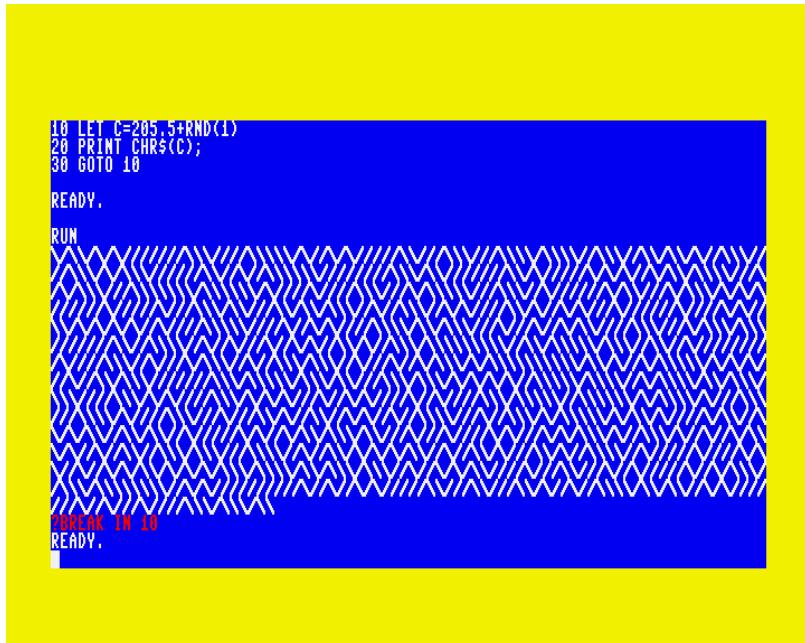
Now, I promised a maze, so I better give you one. We can use this **RND(1)** to pick between these two symbols. The first one has a character code of 205, and the second one conveniently 206. This means that if we add the result of **RND(1)** to 205.5, we will get a number between 205.5 and 206.5. Half the time it will be 205.*something*, and the other half of the time it will be 206.*something*. We can use this to print one or the other characters by using the **CHR\$(C)** function that returns the character corresponding to the number we put between the brackets. This means we can do something like:

```
LET C = 205.5+RND(1)
PRINT CHR$(C);
```

This will print one or the other of these symbols each time. We could use this already to print the maze by doing this over and over, making a loop. We could use **FOR** and **NEXT**. But in this case, we want it to go forever, that is, each time the programme gets to the end, we want it to go to the start again. The people who created BASIC really weren't very creative, so the command to do this is called **GOTO**. You put the number of the line that you want to be executed next after it, e.g., **GOTO 1**. We can use this to write our little maze programme so that it will run continuously:

```
10 LET C = 205.5+RND(1)
20 PRINT CHR$(C);
30 GOTO 10
```

If you **RUN** this programme, it will start drawing a maze forever, that looks like the screenshot below. You can stop it at any time by pressing the **RUN STOP** key, or you can pause it by pressing the **NO SCROLL** key, and unpause it by pressing the **NO SCROLL** key again. If you press the **RUN STOP** key, the computer will tell you where it was up to at the time. In the case of the screenshot below, it was working on line 10:



That works nicely, and draws a very famous maze [?]. We can, however, make the programme smaller. We don't need to put the result of the calculation of which symbol to display on a separate line. We can put the calculation directly into brackets for the **CHR\$(C)** function:

```
10 PRINT CHR$(205.5+RND(1));
20 GOTO 10
```

And we can use what we learnt about the : (colon) symbol, and put the **GOTO** command onto the same line as the **PRINT** command:

```
10 PRINT CHR$(205.5+RND(1));: GOTO 10
```

Can you see how there are often many ways to get the same effect from a programme? This is quite normal. For complex programmes, there are many, many ways to get the

same function. This is one of the areas in computer programming where you can be very creative.

But back to the topic of randomness. It's all well and good using these random numbers between 0 and 1 for drawing a maze, but it's a bit tricky to ask people to get a really long decimal. If we want a number in the range 1 to 100, we can multiply what we get from **RND(1)** by 100. If we do that, it gets a bit better, but we will still get numbers like **55.0304651**, **30.3140154**, **60.2505497** and **.759229916**.

That's closer, but we really want to get rid of those fractional parts. That is, we want whole numbers or *integers*. BASIC has the **INT()** function that works like the **RND(1)** function, except that whatever number you put in the brackets, it will return just the whole part of that. So for example **INT(2.18787)** will return the value **2**. As I said just now, it chops off the fractional part, that is, it always rounds down. So even if we do **INT(2.9999999)** the result will still be **2**, not **3**. This means that if we multiply the result of **RND(1)** by 100, we will get a number in the range of 0 - 99, not 1 - 100. This is nice and easy to fix: We can just add 1 to the result. So to generate an integer, that is a whole number, that is between 1 and 100 inclusive, we can do something like:

```
PRINT INT(RND(1)*100) + 1
```

That looks much better. So let's type in our "guess the number" programme again. But this time, let's replace the place where we set our secret number to the number 23, to instead set it to a random integer between 1 and 100. Don't peek at the solution just yet. Have a think about how we can use the above to set **SN** to a random integer between 1 and 100. Once you have your guess ready, have a look what I came up with below. You might have made a different programme that can do the same job. That's quite fine, too!

```
10 SN=INT(RND(1)*100)+1
20 PRINT " "
30 FOR I = 1 TO 10 STEP 1
40 PRINT "GUESS THE NUMBER BETWEEN 1 AND 100"
50 INPUT "WHAT IS YOUR GUESS"; G
60 IF G<SN THEN PRINT "MY NUMBER IS BIGGER"
70 IF G>SN THEN PRINT "MY NUMBER IS SMALLER"
80 IF G=SN THEN PRINT "CONGRATULATIONS! YOU GUESSED MY NUMBER!": END
90 NEXT I
100 PRINT "SORRY, YOU HAVE RUN OUT OF GUESSES"
```

Now we don't have to worry about someone guessing the number, and we don't need someone else to pick the number for us. This makes the programme much more fun to play. Can you beat it?

## **Exercises to try**

### **1. Can you make the maze programme make different mazes?**

The maze programme currently displays equal numbers of  $\blacksquare$  and  $\blacksquare$ . Can you change the programme to print twice as many of one than the other? How does the maze look then?

Clue: We used **205.5** so that when we add a random number between 0 and 1, we end up with **205.something** half the time and **206.something** the other half of the time. If you reduce **205.5** towards **205**, or increase it towards **206** you will change the relative proportion of each character that appears.

### **2. Can you modify the “guess my number” programme to choose a number between 1 and 10?**

At the moment, the programme picks a number between 1 and 100. Modify the programme so that it picks a number from a different range. Don’t forget to update the message printed to the user. Do they still need 10 guesses? Change the maximum number of guesses they get before losing to a more suitable amount.

Clue: You will need to modify the line that sets **SM**, as well as the **PRINT** message that gives instruction to the user.

### **3. Set the screen, border and text colour to random colours**

Modify either the maze or “guess my number” programme to use random colours. How might you make sure that the text is always visible?

Clue: Use the **FOREGROUND**, **BACKGROUND** and **BORDER** commands to set the colours. Use colour numbers between 0 and 15, inclusive. You can put a calculation at the end of these commands in place of a simple number.

Clue: To make sure you don’t set the text colour to the same as the background, you might like to calculate which background colour you wish to use and keep it in one variable, and then calculate the text colour to use and store it in a different variable. If the two variables have the same number, then you need to change one of them.

### **4. Make the “guess my number” programme randomly choose between two different greeting messages when it starts.**

The “guess my number” programme currently always prints the same message every time it starts. Modify it so that it prints one of two possible messages each time.

Clue: Use **RND(1)** to obtain a random number. If that number is less than some threshold, print the first message, else print the second message.

Clue: It might be easier, if you store the random number in a variable, so that you can use two **IF** statements to decide whether to print each message.

*Clue: If you use < (less than) as the relational operator in one of the IF statements, you will need to use the opposite in the other one. The opposite of less than is greater than or equal to.*



# 8

## CHAPTER

# Text Processing

- **Characters and Strings**
- **String Literals**
- **String Variables**
- **String Statements**
- **Simple Formatting**
- **Sample Programs**



# CHARACTERS AND STRINGS

Representing textual information in the form of printable letters, numbers and symbols is a common requirement of many computer programs. The need for text arises in word processing applications and word games. It is also required in natural language processing and text-based adventure games, both of which need to understand the input. Understanding text input is called *parsing*. In short, text processing is used everywhere. In order to input, output and manipulate such information, we must introduce two key concepts: characters and strings.

Characters can be printable or non-printable. A character most often represents a single, primitive element of printable text which may be displayed on the screen via the statement **PRINT**. It is most common and most natural to think of a character as representing a letter of an alphabet. A character might, for example, be any of the uppercase letters 'A' to 'Z', or any of the lowercase letters 'a' to 'z'. However, characters can also represent commonly used symbols such as punctuation marks or currency symbols. Indeed, characters can also represent the decimal digits, '0' to '9'. It is worth noting that this refers to the text-based representation of the numerals 0 to 9 as printable symbols as opposed to their numeric counterparts. In addition, the MEGA65 provides an extensive range of special symbols that can be used together for games, for drawing fancy borders or art. Besides displaying information, such symbols can create simple yet intriguing visual patterns. For convenience, these special symbols appear on the front sides of the MEGA65's keys.

A character can also be non-printable. Using such characters (in a **PRINT** statement) can activate certain behaviors or cause certain modes to become active, such as the switching of all text on the screen to lowercase or setting the foreground color to orange. Other non-printable characters might represent a carriage return or clear the screen.

For a complete catalog of available characters, refer to Appendix C PETSCII Codes and **CHR\$**. The table lists the characters that correspond to a given code number. The code number must be supplied as an argument to the statement **CHR\$** which, when combined with the **PRINT** statement, outputs the respective characters to the screen.

Here's an example of printing the exclamation mark using a character code:

```
PRINT CHR$(33)  
!
```

Note that the '!' is actually visible on the display because it is a printable character.

Here's an example of changing the foreground color to white using character codes:

```
PRINT CHR$(5)
```

Although no character is output, all subsequent printable characters displayed will be colored white.

Sometimes it can be useful to do the conversion in reverse: from a character to its code number. To do this, a single character must be supplied as an argument to the statement **ASC** within quotation marks which, when combined with the **PRINT** statement, outputs the respective code number to the screen in decimal.

Here's an example of obtaining the code number for the exclamation mark.

```
PRINT ASC("!")
33
```

And here's an example of obtaining the code number for the exclamation mark and storing it in an integer variable:

```
A% = ASC("!")
```

Although we could output individual characters repeatedly by using **CHR\$** it would be tedious to do this all the time.

The concept of a string is needed because it embodies the idea of a contiguous block of text. Thus, a string can contain multiple printable and/or multiple non-printable characters in any combination. A string can potentially be empty and contain no characters at all. To write a string we enclose the characters inside quotation marks. So "HELLO WORLD!" is an example of a string literal.

```
PRINT "HELLO WORLD!"
HELLO WORLD!
```

All strings have a property called length which is how many printable and non-printable characters there are present in that string. The length can be as low as 0 (the empty string) or as high as 255. Attempting to create a string with a length in excess of 255 characters results in a **STRING TOO LONG ERROR**.

```
PRINT LEN("HELLO WORLD!")
12
```

```
PRINT LEN("")
0
```

It is possible to create variables specifically for strings. All such string variables have names that begin with a leading alphabetic character, have an optional second character

that is alphanumeric, and end with a \$ sign. Once given a value, they can be used with **PRINT**.

```
AB$ = "HELLO WORLD!": PRINT AB$  
HELLO WORLD!
```

```
A1$ = "HELLO WORLD!": PRINT LEN(A1$)  
12
```

## STRING LITERALS

String literals can be joined with one or more other such string literals to form a compound string. This process is called *concatenation*. To concatenate two or more string literals, use the + operator to chain them together.

Here are some examples:

```
PRINT ("SECOND" + "HAND")  
SECONDHAND↑↑I
```

```
PRINT ("COUNTER" + "CLOCK" + "WISE")  
COUNTERCLOCKWISE
```

Sometimes punctuation or spaces may be required to make the final output appear correctly formatted, as in the following example.

```
PRINT ("FRUIT: " + "APPLE, " + "PEAR AND " + "RASPBERRY.")  
FRUIT: APPLE, PEAR AND RASPBERRY.
```

## STRING VARIABLES

Concatenation is more commonly used with string variables combined with string literals. For example, in a text-based adventure game you might want to list some exits such as north or south. Because these exits will vary depending on the location you are currently at it would make sense to use variables for the exits themselves and use concatenation with literals such as commas, spaces and full stops to format the output appropriately.

```
A$ = "PEA": B$ = "NUT": PRINT (A$ + B$ + "BUTTER")
PEANUTBUTTER
```

It is also possible to use strings as the parameters of **DATA** statements, to be read later, using the **READ** statement. The following example also demonstrates that arrays can hold strings too.

```
10 DIM A$(6)
20 PRINT "RAINBOW COLORS: ";
30 FOR I = 0 TO 5
40 : READ A$(I): PRINT (A$(I) + ", ");
50 NEXT I
60 READ A$(I): PRINT ("AND " + A$(I) + ".")
70 DATA "RED", "ORANGE", "YELLOW", "GREEN", "BLUE", "INDIGO", "VIOLET"
```

It is common for string data or single-character data to come directly from user input. When the user types some text, that text will often need to be parsed or printed back to the screen. In general, there are three main ways that this can be done: via the **GET** statement, via the **GETKEY** statement or via the **INPUT** statement.

All three statements have different behaviours, and it's important to understand how each one operates by contrasting and comparing them.

The **GET** statement is useful for storing the current keypress in a variable. The program does not wait for a keypress: it continues executing the next statement immediately. For this reason it is sometimes important to place the **GET** statement inside some kind of loop—the loop is to be exited only when a valid keypress is detected. If the variable to **GET** is a string variable and no keypress is detected, then that string variable is set to equal an empty string.

```
10 GET A$: REM DO NOT WAIT FOR A KEYPRESS--READ ANY KEYPRESS INTO THE VARIABLE
20 PRINT A$: IF (A$ = "Y" OR A$ = "N") THEN END
30 GOTO 10
```

The **GETKEY** statement is also useful for storing the current keypress in a variable. In contrast to the **GET** statement, the **GETKEY** statement, when executed, does wait for a single keypress before it continues executing the next statement.

```
10 GETKEY A$: REM WAIT FOR A KEYPRESS--PAUSE AND READ IT INTO THE VARIABLE
20 PRINT A$: IF (A$ = "Y" OR A$ = "N") THEN END
30 GOTO 10
```

While **GET** and **GETKEY** are fine for reading single characters, the **INPUT** statement is useful for reading in entire strings—that is, zero or more characters at a time.

When the **INPUT** statement is used with a comma and a variable, the prompt string is displayed normally with a cursor that permits the user to type in some text. When the **INPUT** statement is used with a semicolon and a variable, the prompt string is displayed with a question mark appended and a cursor that permits the user to type in some text.

```
10 INPUT "ENTER YOUR NAME", A$: REM NOT A QUESTION  
20 PRINT ("HELLO " + A$)
```

```
10 INPUT "WHAT IS YOUR NAME"; A$: REM A QUESTION  
20 PRINT ("HELLO " + A$)
```

In either case, pressing **RETURN** will complete the text entry—the text entered will be stored in the given variable. Note that if the string variable is already equal to some string and **RETURN** is pressed without entering in new data, then the old string value currently stored in the variable is retained.

## STRING STATEMENTS

There are three commonly-used string manipulation commands: **MID\$**, **LEFT\$** and **RIGHT\$**. These are good for isolating substrings, including individual characters.

The following program asks for an input string and then prints all left substrings.

```
10 INPUT "ENTER A WORD:", A$  
20 PRINT "ALL LEFT SUBSTRINGS ARE:"  
30 FOR I = 0 TO LEN(A$)  
40 : PRINT LEFT$(A$, I)  
50 NEXT I
```

The following program asks for an input string and then prints all right substrings.

```
10 INPUT "ENTER A WORD:", A$  
20 PRINT "ALL RIGHT SUBSTRINGS ARE:"  
30 FOR I = 0 TO LEN(A$)  
40 : PRINT RIGHT$(A$, I)  
50 NEXT I
```

The following program ask for an input string consisting of a first name following by a space followed by a last name. It then outputs the initial letters of both names.

```
10 INPUT "ENTER A FIRST NAME, A SPACE AND A LAST NAME:", A$  
20 N = -1  
30 FOR I = 1 TO LEN(A$)  
40 : IF (MID$(A$, I, 1) = " ") THEN N = I: GOTO 60  
50 NEXT I  
60 IF (N = -1) THEN GOTO 10  
70 PRINT "INITIALS ARE: "; MID$(A$, 1, 1)+"."+MID$(A$, N + 1, 1)+". "
```

## SIMPLE FORMATTING

### Suppressing New Lines

When using the **PRINT** statement in a program, the default behaviour is to output the string and then move to the next line. To stop the behaviour of automatically moving to the next line, simply append a ; (semicolon) after the end of the string. Contrast lines 10, 20 and 30 in the following program.

```
10 PRINT "THIS A SINGLE LINE OF TEXT": REM A NEW LINE IS ADDED AT THE END  
20 PRINT "THE SECOND LINE"; : REM A NEW LINE IS SUPPRESSED  
30 PRINT " USES A SEMICOLON" : REM A NEW LINE IS ADDED AT THE END
```

### Automatic Tab Stops

Sometimes it can be convenient to use the **PRINT** statement to output information neatly into columns. This can be done by appending a , (comma) after the end of the string. Consider the following example program.

```
10 PRINT "TEXT 1", "TEXT 2", "TEXT 3", "TEXT 4"
```

Note that each tab stop is 10 characters apart. So TEXT 1 begins at column 0, TEXT 2 begins at column 10, TEXT 3 begins at column 20, and TEXT 4 begins at column 30.

### Tabs Stops and Spacing

When printing text on the screen, it is often necessary to format text by using spaces and tabs. Two commands come in handy here: **SPC** and **TAB**.

The command **SPC(5)**, for example, moves five characters to the right. Any intervening characters that lie between the current cursor position and the position five characters to the right are left unchanged.

The command **TAB(20)**, for example, moves to column 20 by subtracting the cursor's current position away from twenty and then moving that number of characters to the right. If the cursor's initial position is to the right of column 20 then the command does nothing. This command can often be used to make text line up neatly into columns.

## SAMPLE PROGRAMS

We conclude with some examples.

### Palindromes

A *palindrome* is a word or phrase or number that reads the same forwards as it does backwards. Some examples are: CIVIC, LEVEL, RADAR, MADAM and 1234321. The following program reverses the input text and then determines whether the original phrase is equal to the reversed phrase.

```
10 REM *** PALINDROMES ***
20 INPUT "ENTER SOME TEXT: ", A$
30 B$ = ""
40 FOR I = 1 TO LEN(A$)
50 : B$ = MID$(A$, I, 1) + B$
60 NEXT I
70 IF (A$ = B$) THEN PRINT (A$ + " IS A PALINDROME"): ELSE PRINT (A$ + " IS NOT A PALIN
80 GOTO 20
```

### Simple Ciphers

We now look at three simple examples of scrambling and unscrambling English language text messages. This scrambling and unscrambling process is the study of *cryptography* and is used to keep information secure so that it can't be read by others except for those privileged to know the cipher's method and secret key.

The process of scrambling a given message is called *encryption*. The ordinary, readable unscrambled text is called *plaintext*. Encrypting plaintext results in a scrambled message. This scrambled text is called *ciphertext*. The process of unscrambling the ciphertext is called *decryption*. Decrypting the ciphertext results in an unscrambled message—the plaintext.

Suppose that we were to encrypt some plaintext and then send the resulting ciphertext to a friend. Provided that the friend knows the method and secret key used to scramble the message, they could then decrypt the ciphertext and would be able to recover and read our original plaintext message.

If someone else attempts to read the ciphertext using the wrong method and/or the wrong secret key, the resulting text will be unintelligible.

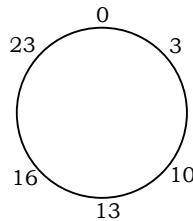
The cryptographic systems we describe here are very simple. Obviously, they shouldn't be used today because they are easily broken by techniques of cryptanalysis. Nevertheless, they illustrate some basic techniques and show how we might structure a sample program.

We investigate three ciphers. These are the ROT13 cipher, the Caesar Cipher and the Atbash Cipher. These are part of a group of ciphers known as *affine ciphers*.

Mathematically, it is useful to think of the letters of the English alphabet as numbered. A is 0, B is 1 and so, with Z being equal to 25.

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

A key mathematical component of a cryptographic system is *modular arithmetic*, sometimes casually referred to as "clock arithmetic" because the numbers begin at zero and increase until they reach an upper limit, at which point they wrap around back to zero again, much like a circle. In our case, since there are 26 letters in the English alphabet, we use modulo 26 arithmetic—our letters are numbered from 0 to 25.



To reduce a given number using modulo 26 we can use the following function:

$$f(x) = x - \left\lfloor \frac{x}{26} \right\rfloor \times 26$$

This says that to obtain the value of a number  $x$  using modulo 26 we first divide  $x$  by 26 and round down, which gives us the number of times we went around the circle. We then multiply this result by 26 again and subtract this from  $x$ . The final result is the remainder left over and will always be a value between 0 and 25.

As an example, the number 28 in modulo 26 is equal to 2:

$$f(28) = 28 - \left\lfloor \frac{28}{26} \right\rfloor \times 26 = 28 - 1 \times 26 = 2$$

The program at the end of this chapter makes use of this formula by defining a corresponding function at line 30:

```
DEF FN F(X)=X-INT(X/26)*26
```

**ROT13:** When we encrypt each plaintext letter we move forward 13 places. So the plaintext letter A becomes the ciphertext letter N, B becomes O, with latter letters "wrapping around" back to the beginning of the alphabet. Thus, the plaintext letter Z becomes the ciphertext letter M. This covers encryption. To decrypt each ciphertext letter we simply repeat the process by moving forward 13 places again, which brings us full circle, back to where we started. Thus, a ciphertext letter N becomes the plaintext letter A.

We can see this visually as a mapping in the form of a table:

English Plaintext	A	B	C	D	E	F	G	H	I	J	K	L	M
ROT13 Ciphertext	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

English Plaintext	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
ROT13 Ciphertext	A	B	C	D	E	F	G	H	I	J	K	L	M

To encrypt using ROT13, find the plaintext letter in the top row and move down to the bottom row to find the corresponding ciphertext letter. To decrypt using ROT13, find the ciphertext letter in the bottom row and move up to the top row to find the corresponding plaintext letter.

If we consider the ROT13 cipher from a mathematical standpoint, we can see that to both encrypt and decrypt we simply add 13 to the numerical value of a plaintext or ciphertext letter and reduce it using modulo 26. This gives us a new number between 0 and 25 which corresponds to the encrypted or decrypted letter. Function  $E_{ROT13}$  is the encryption function. It accepts the value of a plaintext letter  $x$  as an argument and returns the value of the ciphertext letter as a result. Function  $D_{ROT13}$  is the decryption function. It accepts the value of a ciphertext letter  $x$  as an argument and returns the value of the plaintext letter as a result.

$$E_{ROT13}(x) = (x + 13) \bmod 26$$

$$D_{ROT13}(x) = (x + 13) \bmod 26$$

Notice that the definitions of both the encryption and decryption functions are, in this case, exactly the same.

**Atbash:** Atbash is an ancient technique used to encrypt the 22-letter Hebrew alphabet, but we can apply the same logic to encrypt the 26-letter English alphabet. To encrypt a letter using Atbash we need to consider the English alphabet written backwards. So encrypting the plaintext letter A becomes the ciphertext letter Z, B becomes Y, C becomes X and so on. Decrypting the ciphertext works the same way: the ciphertext letter A becomes the plaintext letter Z, B becomes Y, C becomes X and so on.

We can see this visually as a mapping in the form of a table:

English Plaintext	A	B	C	D	E	F	G	H	I	J	K	L	M
Atbash Ciphertext	Z	Y	X	W	V	U	T	S	R	Q	P	O	N

English Plaintext	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Atbash Ciphertext	M	L	K	J	I	H	G	F	E	D	C	B	A

To encrypt using Atbash, find the plaintext letter in the top row and move down to the bottom row to find the corresponding ciphertext letter. To decrypt using Atbash, find the ciphertext letter in the bottom row and move up to the top row to find the corresponding plaintext letter.

If we consider the Atbash cipher from a mathematical standpoint, we can see that to encrypt and decrypt, we need to multiply by 25 and then add 25 to the numerical value of the plaintext or ciphertext and reduce it using modulo 26. This gives us a new number between 0 and 25 which corresponds to the encrypted or decrypted letter. Function  $E_{Atbash}$  is the encryption function. It accepts the value of a plaintext letter  $x$  as an argument and returns the value of the ciphertext letter as a result. Function  $D_{Atbash}$  is the decryption function. It accepts the value of a ciphertext letter  $x$  as an argument and returns the value of the plaintext letter as a result.

$$E_{Atbash}(x) = (25 \times x + 25) \bmod 26$$

$$D_{Atbash}(x) = (25 \times x + 25) \bmod 26$$

Notice that the definitions of both the encryption and decryption functions are, in this case, exactly the same.

**Caesar:** The Caesar cipher is also an ancient technique used encrypt and decrypt messages. To encrypt a letter using the Caesar cipher we move three positions forward. So encrypting the plaintext letter A becomes the ciphertext letter D, B becomes E, C becomes

F and so on. Decrypting the ciphertext works the opposite way. Instead of moving forward, we move three positions backward. The ciphertext letter A becomes the plaintext letter X, B becomes Y, C becomes Z and so on.

We can see this visually as a mapping in the form of a table:

English Plaintext	A	B	C	D	E	F	G	H	I	J	K	L	M
Caesar Ciphertext	D	E	F	G	H	I	J	K	L	M	N	O	P
English Plaintext	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Caesar Ciphertext	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

To encrypt using the Caesar cipher, find the plaintext letter in the top row and move down to the bottom row to find the corresponding ciphertext letter. To decrypt using the Caesar cipher, find the ciphertext letter in the bottom row and move up to the top row to find the corresponding plaintext letter.

If we consider the Caesar cipher from a mathematical standpoint, we can see that to encrypt, we need to add 3 to the numerical value of the plaintext and reduce it using modulo 26. This gives us a new number between 0 and 25 which corresponds to the encrypted letter. To decrypt, we need to subtract 3 from the numerical value of the ciphertext and reduce it modulo 26. This gives us a new number between 0 and 25 which corresponds to the decrypted letter.

Function  $E_{Caesar}$  is the encryption function. It accepts the value of a plaintext letter  $x$  as an argument and returns the value of the ciphertext letter as a result. Function  $D_{Caesar}$  is the decryption function. It accepts the value of a ciphertext letter  $x$  as an argument and returns the value of the plaintext letter as a result.

$$E_{Caesar}(x) = (x + 3) \bmod 26$$

$$D_{Caesar}(x) = (x - 3) \bmod 26$$

Notice that the definitions of both the encryption and decryption functions are, in this case, different.

We can generalise all three of the above methods by stating that they use the following encryption and decryption functions:

$$E(x) = (A_1x + B_1) \bmod 26$$

$$D(x) = (A_2x + B_2) \bmod 26$$

Here,  $A_1$ ,  $A_2$ ,  $B_1$  and  $B_2$  are constants and put together they comprise the *encryption key* for an affine cipher.

Running the following program displays a text menu. The user can choose to encrypt or decrypt a string, or quit the program. You can practice typing in a plaintext phrase to encrypt and then decrypt the ciphertext phrase to retrieve the original plaintext.

A good sample text string for testing a cipher is:

**THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG**

This text string, which is 43 characters long, contains 8 spaces and 35 alphabetic characters. Every character of the alphabet occurs at least once in this string, so encrypting and decrypting with it checks that every letter is transformed as expected.

Encrypting the above text string using the ROT13 cipher yields:

**GUR DHVPX OEBJA SBK WHZCF BIRE GUR YNML QBT**

Encrypting the above text string using the Atbash cipher yields:

**GSV JFRXP YILDM ULC QFNKH LEVI GSV OZAB WLT**

Encrypting the above text string using the Caesar cipher yields:

**WKH TXLFN EURZQ IRA MXPSV RYHU WKH ODCB GRJ**

```
10 REM *** CRYPTOGRAPHY ***
20 POKE 0,65: PRINT CHR$(142): PRINT CHR$(147)
30 DEF FN F(X)=X-INT(X/26)*26
40 C$="" P$=""
50 PRINT "SELECT AN OPTION (E, D OR Q)": PRINT
60 PRINT "(SPACE*3)[E] ENCRYPT PLAINTEXT": PRINT
70 PRINT "(SPACE*3)[D] DECRYPT CIPHERTEXT": PRINT
80 PRINT "(SPACE*3)[Q] QUIT": PRINT
90 GET $$
100 IF ($$=="Q") THEN END
110 IF ($$=="E") THEN GOSUB 150: GOTO 40
120 IF ($$=="D") THEN GOSUB 270: GOTO 40
130 GOTO 90
140 REM ENCRYPT
150 INPUT "ENTER PLAINTEXT MESSAGE TO ENCRYPT: ", P$
160 IF P$="" THEN GOTO 150
170 M$=P$: GOSUB 390
180 IF (V=0) THEN GOSUB 460: GOTO 150
190 A=1:B=3
200 FOR I=1 TO LEN(P$)
210 : L$ = MID$(P$,I,1)
220 : IF (L$=" ") THEN C$=C$+" ": ELSE C$=C$+CHR$(65+(FN F(A*(ASC(L$)-65)+B)))
230 NEXT I
240 PRINT: PRINT "(REVERSE ON)ENCRYPTED CIPHERTEXT:(REVERSE OFF)", C$: PRINT
250 RETURN
260 REM DECRYPT
270 INPUT "ENTER CIPHERTEXT MESSAGE TO DECRVPT: ", C$
280 IF C$="" THEN GOTO 270
290 M$=C$: GOSUB 390
300 IF (V=0) THEN GOSUB 460: GOTO 270
310 A=1: B=-3
320 FOR I=1 TO LEN(C$)
330 : L$ = MID$(C$,I,1)
340 : IF (L$=" ") THEN P$=P$+" ": ELSE P$=P$+CHR$(65+(FN F(A*(ASC(L$)-65)+B)))
350 NEXT I
360 PRINT: PRINT "(REVERSE ON)DECRYPTED PLAINTEXT:(REVERSE OFF)", P$: PRINT
370 RETURN
380 REM VALIDATE
390 V = 1
400 FOR I=1 TO LEN(M$)
410 : L$ = MID$(M$,I,1)
420 : IF NOT (((L$ >= "A") AND (L$ <= "Z")) OR (L$=" ")) THEN V = 0
430 NEXT I
440 RETURN
450 REM ERROR MESSAGE
460 PRINT: PRINT "USE LETTERS AND SPACES ONLY": PRINT
470 RETURN††I
```

If you wish to use the ROT13 cipher ensure that the following lines are changed:

```
190 A=1: B=13  
310 A=1: B=13
```

If you wish to use the Atbash cipher ensure that the following lines are changed:

```
190 A=25: B=25  
310 A=25: B=25
```

If you wish to use the Caesar cipher ensure that the following lines are changed:

```
190 A=1: B=3  
310 A=1: B=-3
```

The program listing, as written, uses the Caesar cipher by default.

# 9

## 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 familiar with 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, the MAP instruction and the KEY register that allows you to change the mode of operation of the computer, as well 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 available to programmes written in assembly language / machine code. 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 switches from C64 mode to C65 mode. 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. For more information about which registers are hidden and visible in each of the VIC-II, VIC-III and VIC-IV IO modes refer to Appendix F.

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 un-hide 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 un-hide 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 un-hidden 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 any 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 try the examples from above, the colours won't change 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 been performed, but the second one hasn't yet.)

Again, you won't see any immediate difference, just like when un-hiding 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 we can 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. You might be 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 J.

## 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, 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: POKE0,123: 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, DECC("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 G.

# PART III

## SOUND AND GRAPHICS



# PART IV

# HARDWARE



# 10

## CHAPTER

### 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

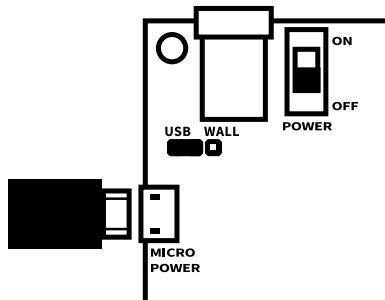
You can build your own MEGA65-compatible computer by using a Nexys4DDR (Nexys A7) FPGA development board. This appendix describes the process to setup a Nexys4DDR board for this purpose. The older non-DDR Nexys4 board is also supported, and the instructions are the same, except that you must use a bitstream 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

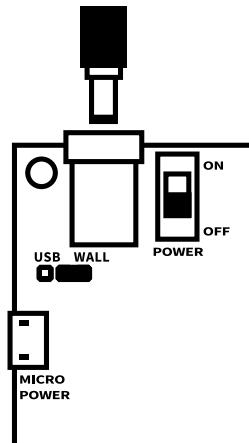
The Nexys4DDR board can be powered in two ways: using an external power supply, or from a standard USB port.

## Micro-USB Power



Connect your micro-usb cable to a USB port on a USB charger or PC to provide power. Connect the other end to the Nexys4DDR's micro-usb connector. Place the JP3 jumper on pins 1 and 2 to select USB power. Use the switch to turn on the Nexys4DDR.

## External Power Supply



The MEGA65 core can consume a lot of power, and a standard USB port could potentially be too little for the Nexys4DDR board. In particular, writing to the SD card might hang or perform odd behaviour. Therefore you should consider 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. The chosen power supply should be center positive, 2.1mm internal diameter plug, and should deliver 4.5VDC to 5.5VDC rated at least 1 Amp.

Connect the power supply cable to the supply plug of the Nexys4DDR. Place the JP3 jumper on pins 2 and 3 to select WALL power. Use the switch to turn on the Nexys4DDR.

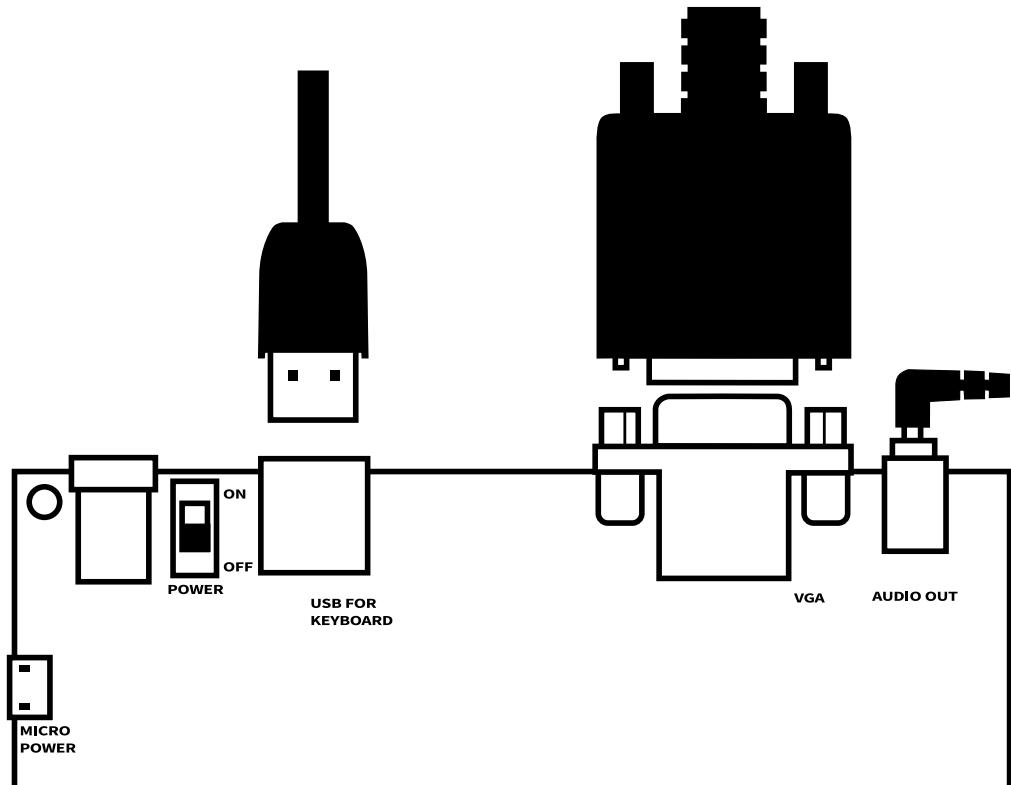
## **Other Jumpers and Switches**

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

- JP1 – USB/SD
- JP2 – SD

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

## Connections and Peripherals



A USB keyboard can be connected to the USB port. Only a keyboard that lacks 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.

You can connect a VGA monitor to the VGA port.

The mono audio-out jack can be connected to the line-in of an amplifier.

### Onboard buttons

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

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 file-system. 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 **ALT** key 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.

For more detailed information on preparing and configuring your MEGA65, please refer to Chapter 4.

## 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 **\$VSS5552** in C64 mode to switch to C65 mode.

- Type **0064** 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 bit-wise 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 DOPEN# 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 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 **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 bank-number**

**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 (Appendix F) for details.

**Remarks:** A value > 127 selects memory mapped I/O. The default value for the bank number is 128. This configuration has RAM from \$0000 to \$1FFF and BASIC ROM's, KERNEL ROM's and I/O from \$2000 to \$FFFF.

**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

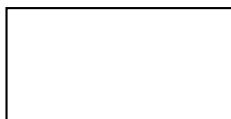
**Format:** **BOX X0,Y0, X1,Y1, X2,Y2, X3,Y3, SOLID**

**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.

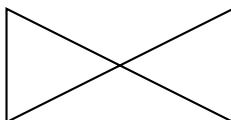
**Remarks:** A quadrangle is a geometric figure with four sides and four angles. A box is a special form of a quadrangle, with all four angles at 90 degrees. Rhomboids, kites and parallelograms are special forms too. So the name of this command is misleading, because it can be used to draw all kind of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

**Example:** Using **BOX**

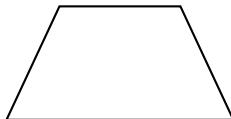
```
BOX 0,0, 160,0, 160,80, 0,80
```



```
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 tokenised 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 384,196, 1,1,2, "MEGA65"
```

will print the text "MEGA65" 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$(27)
30 PRINT QUOTE$;"MEGA65";QUOTE$ : REM PRINT "MEGA65"
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 information 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 PRIN#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 run-time 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 initialised at the next usage.

**Example:** Using **CLR**

```
10 A=5: P$="MEGA65"
20 CLR
30 PRINT A;P$  
  
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 1,0H5
30 SPRITE 2,1 : MOVSPR 2,120,100 : MOVSPR 2,180H5
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 drive] 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. (**FS\$**)

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 initialises 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 "MEGA65",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 = π / 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: (**FSS\$**)

**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 AX(8) :REM ARRAY OF 9 ELEMENTS
20 DIM XX(2,3) :REM ARRAY OF 3x4 = 12 ELEMENTS
30 FOR I=0 TO 8:AX(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 "I8" :REM INITIALISE DISK IN DRIVE 8  
DISK "U8>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 re-linked 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 (ignored for fill mode)

**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 organised in 256 banks of 64 K.

**Example:** Using **DMA**

```
DMA 0, 80*25, 2048, 0, 0, 4 :REM SAVE SCREEN TO $00000 BANK 4  
DMA 3, 80*25, 32, 0, 2048, 0 :REM FILL SCREEN WITH BLANKS  
DMA 0, 80*25, 0, 4, 2048, 0 :REM RESTORE SCREEN FROM $00000 BANK 4  
DMA 2, 80, 2048, 0, 2048+80, 0 :REM SWAP CONTENTS OF LINE 1 & 2 OF SCREEN
```

# **D MODE**

**Token:** \$FE \$35

**Format:** **D MODE jam,complement,inverse,stencil,style,thick**

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

<b>jam</b>	0 - 1
<b>complement</b>	0 - 1
<b>inverse</b>	0 - 1
<b>stencil</b>	0 - 1
<b>style</b>	0 - 3
<b>thick</b>	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 IX=0 : REM INTEGER LOOP 1 -> 100  
20 DO IX=IX+1  
30 LOOP WHILE IX < 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 relative 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(UNK)  
DOPEN#3,"USER FILE,U"  
DOPEN#2,"DATA BASE",L240  
DOPEN#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.

<b>type</b>	0 - 63
<b>number</b>	1 - 4
<b>pattern</b>	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 erasing 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 WITH "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 sound filter.

**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 tokenised 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 initialise 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 GETH2,D$,D$             :REM DISCARD LOAD ADDRESS
25 DO                      :REM LINE LOOP
30 : GETH2,D$,D$           :REM DISCARD LINE LINK
35 : IF ST THEN EXIT       :REM END-OF-FILE
40 : GETH2,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. Use **\$VS59552** to switch back to C65 mode.

**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 run-time 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)";AS
70 LOOP UNTIL AS="Y" OR AS="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 AS="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 run-time 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)";AS
70 LOOP UNTIL AS="Y" OR AS="N"
80 RETURN
90 *** MAIN PROGRAM ***
100 DOPENH2,"BIG DATA"
110 GOTO 30: IF DD THEN DCLOSEH2:GOTO 60:REM ASK
120 IF AS="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 INITIALISE
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",IB$  
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# channel, 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.
- channel** = 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$(100),A%(100),S$(100):
20 DOPEN#2,"DATA"
30 FOR I=0 TO 100
40 INPUT#2,N$(I),A%(I),S$(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 key code 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 INITIALISE
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$)>? OR A$=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
```

# 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 light pen must be connected to port 1.

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

**LPEN(1)** returns the Y position of the light pen, 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 light pen. The **COLLISION** statement may be used to install an interrupt handler.

**Example:** Using LPEN

```
PRINT LPEN(0),LPEN(1) :REM PRINT LIGHT PEN 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 sub-strings into an existing string.

**string** = a string expression

**index** = start index (0 -> 255)

**n** = length of sub-string (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 recognised 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 MOVSPR 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 initialise 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 * 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
```

# NOT

**Token:** \$A8

**Format:** NOT operand

**Usage:** Performs a bit-wise 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 bit-wise 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 INITIALISE
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 INITIALISE
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 INITIALISE
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     :REM 0 = BLACK
60 PALETTE 1,1, 15, 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 "T9"
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 one 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.

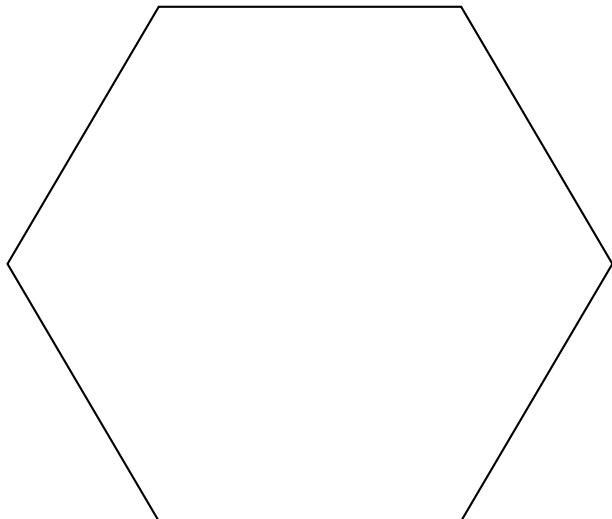
**sides** = sides to draw ( $\leq n$ ).

**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 fire button.

**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.

' can be inserted into large numbers.

'\$' sets the position of the currency symbol.

^^^^ 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 centres 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$ = "MEGA65"
30 PRINT USING "####.##"; X      : REM "12.34"
40 PRINT USING "###"; X        : REM " 12"
50 PRINT USING "##"; A$       : REM "MEGA"
60 PRINT USING "#####"; A$ : REM "MEGA65 "
70 PRINT USING "=#####"; A$ : REM " MEGA65 "
80 PRINT USING "#####)"; A$ : REM " MEGA65"
```

# 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 localisation. 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: multi-colour mode 1
- 3: multi-colour 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 initialises 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$, VE
20 READ NX:FOR I=2 TO NX:READ GL(I):NEXT I
30 PRINT "PROGRAM:";NA$;" 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 "MEGA65",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 DOPENH3,"DATA INDEX"
25 FOR I=1 TO N:INPUTH3,IX(I):NEXT
30 DCLOSEH3
35 DOPENH2,"DATA BASE",L240
40 FOR J=1 TO 10
45 RECORDH2,IX(J)
50 INPUTH2,A$
55 PRINT A$
60 NEXT J
65 DCLOSEH2
```

# **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 601 TO 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 "MEGAS5"
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 FNDI(X) = INT(RND(0)*6)+1 :REM DICE FUNCTION
20 FOR I=1 TO 10
30 PRINT I;FNDI(0)
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 multi-colour sprite colours.

n = 1 : get multi-colour # 1.

n = 2 : get multi-colour # 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 : multi-colour (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 MULTI-COLOUR ?
```

# 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 run-time 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 in-file 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 INITIALISE
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 WITH "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 initialises 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 INITIALISE
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 initialised 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 ON 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 SWEEEPING 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 multi-colour 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 multi-colour 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 1,0H5
30 SPRITE 2,1 : MOVSPr 2,120,100 : MOVSPr 2,180H5
40 FOR I=1 TO 50000:NEXT
50 COLLISTION 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 initialise 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

<b>Token:</b>	\$9E
<b>Format:</b>	<b>SYS address [, areg, xreg, yreg, zreg, sreg]</b>
<b>Usage:</b>	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 <b>JSR address</b> is performed. The called routine should exit with a <b>RTS</b> instruction. Then the register contents will be saved and the execution of the BASIC program continues.
<b>address</b>	= start address of the subroutine.
<b>areg</b>	= variable gets accumulator value.
<b>xreg</b>	= variable gets X register value.
<b>yreg</b>	= variable gets Y register value.
<b>zreg</b>	= variable gets Z register value.
<b>sreg</b>	= variable gets status register value.
The <b>SYS</b> command uses the current bank as set with the <b>BANK</b> command.	
<b>Remarks:</b>	The register values after a <b>SYS</b> call are stored in system memory. This enables the command <b>RREG</b> to retrieve these values.
<b>Remarks:</b>	The system leaves the memory setup in a somewhat strange state, which may or may not be a bug in BASIC 10. If you want to write an assembly routine that can access an entire memory bank, but with IO mapped, you need to set the CPU register \$01, like on the C64, and then also use the 4510's MAP instruction to set the memory map. For BANK 0, you could use the following sequence:

```

; Disable interrupts, because the C65 ROM won't be visible to
; handle them.
SEI
; Make IO visible
LDA #$37
STA $01
; Prepare for MAP operation that resets memory map to BANK 0
LDA #$00
TAX
TAY
TAZ
MAP
; End MAPping sequence
NOP
; Now comes your routine!
loop:
INC $D020
JMP loop

```

- Remarks:** You can alternatively use BANK 128, and place your routine in the lower-half of RAM, e.g., in the unallocated memory area at \$1600 - \$1FFF
- Remarks:** The RREG command can be useful for reading the register values when your routine completes.
- 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 tab-stop.

**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  8.22381268E+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. **(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:** 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 LST",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 IX=0 : REM INTEGER LOOP 1 -> 100
20 DO IX=IX+1
30 LOOP WHILE IX < 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 centres 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$ = "MEGA65"
30 PRINT USING "####.##"; X      : REM "12.34"
40 PRINT USING "###"; X        : REM " 12"
50 PRINT USING "##"; A$       : REM "MEGA"
60 PRINT USING "#####"; A$ : REM "MEGA65 "
70 PRINT USING "=#####"; A$ : REM " MEGA65 "
80 PRINT USING "#####)"; A$ : REM " MEGA65"
```

# 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(@)           :REM PRINT RESULT FOR ARGUMENT P1
```

# 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 "I: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 IX=0 : REM INTEGER LOOP 1 -> 100  
20 DO IX=IX+1  
30 LOOP WHILE IX < 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 CENTRED 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 bit-wise 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

- PETSCII Codes and CHR\$
- Control codes
- Shifted codes
- Escape Sequences



# PETSCII CODES AND CHR\$

You can use the `PRINT CHR$(X)` statement to print a character. Below is the full table of PETSCII codes you can print by index. For example, by using index 65 from the table below as: `PRINT CHR$(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 with the code in the table.

<b>0</b>	<b>22</b>	<b>45</b> -	<b>68</b> D
<b>1</b>	<b>23</b>	<b>46</b> .	<b>69</b> E
<b>2</b> UNDERLINE ON	<b>24</b>	<b>47</b> /	<b>70</b> F
<b>3</b>	<b>25</b>	<b>48</b> 0	<b>71</b> G
<b>4</b>	<b>26</b>	<b>49</b> 1	<b>72</b> H
<b>5</b> WHITE	<b>27</b> ESCAPE	<b>50</b> 2	<b>73</b> I
<b>6</b>	<b>28</b> RED	<b>51</b> 3	<b>74</b> J
<b>7</b> BELL	<b>29</b> →	<b>52</b> 4	<b>75</b> K
<b>8</b> DISABLE  	<b>30</b> GREEN	<b>53</b> 5	<b>76</b> L
<b>9</b> ENABLE  	<b>31</b> BLUE	<b>54</b> 6	<b>77</b> M
<b>10</b> LINEFEED	<b>32</b> <b>SPACE</b>	<b>55</b> 7	<b>78</b> N
<b>11</b>	<b>33</b> !	<b>56</b> 8	<b>79</b> O
<b>12</b>	<b>34</b> "	<b>57</b> 9	<b>80</b> P
<b>13</b> <b>RETURN</b>	<b>35</b> #	<b>58</b> :	<b>81</b> Q
<b>14</b> LOWER CASE	<b>36</b> \$	<b>59</b> ;	<b>82</b> R
<b>15</b> BLINK ON	<b>37</b> %	<b>60</b> <	<b>83</b> S
<b>16</b>	<b>38</b> &	<b>61</b> =	<b>84</b> T
<b>17</b> ↓	<b>39</b> '	<b>62</b> >	<b>85</b> U
<b>18</b> 	<b>40</b> (	<b>63</b> ?	<b>86</b> V
<b>19</b> 	<b>41</b> )	<b>64</b> @	<b>87</b> W
<b>20</b> 	<b>42</b> *	<b>65</b> A	<b>88</b> X
<b>21</b>	<b>43</b> +	<b>66</b> B	<b>89</b> Y
	<b>44</b> ,	<b>67</b> C	<b>90</b> Z

91 [	117 ☐	142 UPPERCASE	166 ☒
92 £	118 ☒	143 BLINK OFF	167 ☐
93 ]	119 ☒	144 BLACK	168 ☒
94 ↑	120 ☒	145 	169 ☒
95 ←	121 ☐	146 	170 ☐
96 ☐	122 ☒	147 	171 ☐
97 ☒	123 ☐	148 	172 ☐
98 ☐	124 ☒	149 BROWN	173 ☒
99 ☐	125 ☐	150 LT. RED	174 ☒
100 ☐	126 π	151 DK. GRAY	175 ☐
101 ☐	127 ☒	152 GRAY	176 ☐
102 ☐	128	153 LT. GREEN	177 ☐
103 ☐	129 ORANGE	154 LT. BLUE	178 ☐
104 ☐	130 UNDERLINE OFF	155 LT. GRAY	179 ☐
105 ☒	131	156 PURPLE	180 ☐
106 ☒	132	157 	181 ☐
107 ☒	133 F1	158 YELLOW	182 ☐
108 ☐	134 F3	159 CYAN	183 ☐
109 ☒	135 F5	160 	184 ☐
110 ☒	136 F7	161 ☐	185 ☐
111 ☐	137 F2	162 ☐	186 ☐
112 ☐	138 F4	163 ☐	187 ☐
113 ☒	139 F6	164 ☐	188 ☐
114 ☐	140 F8	165 ☐	189 ☐
115 ☒	141  	190 ☐	191 ☐

NOTE: Codes for 192 to 223 are the equal to 96-127. Codes 224 to 254 equal to 160-190 and code 255 equal to 126.

# CONTROL CODES

Keyboard Control	Function
<b>CTRL</b> + <b>1</b> to <b>8</b>	Choose from the first range of colours.
<b>CTRL</b> + <b>T</b>	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.
<b>CTRL</b> + <b>Z</b>	Tabs the cursor to the left.
<b>CTRL</b> + <b>E</b>	Restores the colour of the cursor back to the default white.
<b>CTRL</b> + <b>Q</b>	moves the cursor down one line at a time. This is the same function produced by the Cursor Down key.
<b>CTRL</b> + <b>G</b>	produces a bell tone.
<b>CTRL</b> + <b>J</b>	is a line feed and moves the cursor down one row. This is the same function produced by the <b>↓</b> key.
<b>CTRL</b> + <b>U</b>	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.
<b>CTRL</b> + <b>W</b>	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
<b>CTRL</b> + <b>B</b>	turns on underline text mode. Turn off underline mode by pressing <b>ESC</b> then <b>O</b> .
<b>CTRL</b> + <b>N</b>	changes the text case mode from uppercase to lowercase.
<b>CTRL</b> + <b>M</b>	is the carriage return. This is the same function as the <b>RETURN</b> key.
<b>CTRL</b> + <b>]</b>	is the same function as <b>→</b> .
<b>CTRL</b> + <b>I</b>	tabs forward to the right.
<b>CTRL</b> + <b>X</b>	sets or clears the current screen column as a tab position. <b>CTRL</b> + <b>I</b> or <b>Z</b> will jump to all positions set with <b>X</b> . When there are no more tab positions, the cursor will stay at the end of the line with <b>CTRL</b> and <b>I</b> , or move to the start of the line in the case of <b>CTRL</b> and <b>Z</b> .
<b>CTRL</b> + <b>K</b>	locks the uppercase/lowercase mode switch usually performed with <b>Y</b> and <b>SHIFT</b> keys.
<b>CTRL</b> + <b>L</b>	enables the uppercase/lowercase mode switch that is performed with the <b>Y</b> and <b>SHIFT</b> keys.
<b>CTRL</b> + <b>[</b>	is the same as pressing the <b>ESC</b> key.
<b>CTRL</b> + <b>*</b>	enters the Matrix Mode Debugger.

# SHIFTED CODES

Keyboard Control	Function
<b>SHIFT</b> + <b>INST</b>	Insert a character in the current cursor position and move all characters to the right by one position.
<b>SHIFT</b> + <b>HOME</b>	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
<b>X</b>	Clears the screen and toggles between 40 and 80 column modes.
<b>@</b>	Clears the screen starting from the cursor to the end of the screen.
<b>A</b>	Enables the auto-insert mode. Any keys pressed will insert before other characters.
<b>B</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 <b>ESC</b> then <b>T</b> .
<b>C</b>	Disables auto-insert mode, going back to overwrite mode.
<b>D</b>	Deletes the current line and moves other lines up one position.
<b>E</b>	Sets the cursor to non-flashing mode.
<b>F</b>	Sets the cursor to regular flashing mode.
<b>G</b>	Enables the bell which can be sounded using <b>CTRL</b> and <b>G</b> .
<b>H</b>	Disable the bell so that pressing <b>CTRL</b> and <b>G</b> will have no effect.
<b>I</b>	Inserts an empty line in the current cursor position and moves all subsequent lines down one position.

<b>Key</b>	<b>Sequence</b>
<b>J</b>	Moves the cursor to start of current line.
<b>K</b>	Move to end of the last non-white-space character on the current line.
<b>L</b>	Enables scrolling when the cursor down key is pressed at the bottom of the screen.
<b>M</b>	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.
<b>O</b>	Cancels the quote, reverse, underline and flash modes.
<b>P</b>	Erases all characters from the cursor to the start of current line.
<b>Q</b>	Erases all characters from the cursor to the end of current line.
<b>S</b>	Switches the VIC-IV to colour range 16-31. These colours can be accessed with <b>CTRL</b> and keys <b>1</b> to <b>8</b> or <b>M</b> and keys <b>1</b> to <b>8</b> .
<b>T</b>	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 <b>ESC</b> then <b>B</b> .
<b>U</b>	Switches the VIC-IV to colour range 0-15. These colours can be accessed with <b>CTRL</b> and keys <b>1</b> to <b>8</b> or <b>M</b> and keys <b>1</b> to <b>8</b> .

Key	Sequence
<b>V</b>	Scrolls the entire screen up one line.
<b>W</b>	Scrolls the entire screen down one line.
<b>X</b>	Toggles the 40/80 column display. The screen will also clear home.
<b>Y</b>	Set the default tab stops (every 8 spaces) for the entire screen.
<b>Z</b>	Clears all the tab stops. Any tabbing with <b>CTRL</b> and <b>I</b> will move the cursor to the end of the line.
<b>1</b> to <b>8</b>	Choose from the second range of colours.

# D

## APPENDIX

### The MEGA65 Keyboard

- **Hardware Accelerated Keyboard Scanning**
- **Keyboard Theory of Operation**
- **C65 Keyboard Matrix**
- **Synthetic Key Events**
- **Keyboard LED Control**
- **Native Keyboard Matrix**



The MEGA65 has a full mechanical keyboard which is compatible with the C65 and C64 keyboards, and features four distinct cursor keys which work in both C64 and C65 mode, as well as eleven new C65 keys that normally work only in C65 mode.

## HARDWARE ACCELERATED KEYBOARD SCANNING

To make use of the new extended keyboard easier, the MEGA65 features a hardware accelerated keyboard scan circuit, that provides ASCII (not PETSCII!) codes for keys and key-combinations. This makes it very simple to use the full capabilities of the MEGA65's keyboard, including the entry of ASCII symbols such as {, \_, and }, which are not possible to type on a normal C64 and C128 keyboards.

The hardware accelerated keyboard scanner has a buffer of 3 keys, which helps to make it easier to read from the keyboard without having check it too regularly. Further, the hardware accelerated keyboard scanner supports most Latin-1 code-page characters, allowing the entry of many accented characters. These keys are entered by holding down the  key and pressing other keys or key-combinations. The use of ASCII or Latin-1 symbols not present in the PETSCII character set requires the use of a font that contains these symbols, and software which supports them.

The hardware accelerated keyboard scanner is very simple to use: First, make sure that you have the MEGA65 IO context activated, then read memory location \$D610 (decimal 54800). If the register contains zero, no key has been pressed. Otherwise the value will be the ASCII code of the most recent key or key-combination that has been pressed. Reading \$D610 again will continue to read the same value until you POKE any value into \$D610. This clears the key from the input buffer.

The hardware accelerated keyboard scanner also provides a register that indicates which of the modifier keys are currently being held down. This is accessed via the read-only register \$D611 (decimal 54801):

**Bit 0** Right 

**Bit 3** 

**Bit 1** Left 

**Bit 4** 

**Bit 2** 

**Bit 5** 

Note that the hardware accelerated keyboard scanner operates independently of the C64 or C65 KERNAL keyboard scanning routines. That is, the KERNAL will still have any keys that you have entered buffered in the normal way. For assembly language programs the easiest solution to this is to disable interrupts via the SEI instruction. This prevents the KERNAL keyboard scanner from running.

## Latin-1 Keyboard Map

### KEYBOARD THEORY OF OPERATION

The MEGA65 keyboard is a full mechanical keyboard, constructed as a matrix. Every key switch is fitted with a diode, which allows the keyboard hardware to detect when any combination of keys are pressed at the same time. This matrix is scanned by the firmware in the CPLD chip on the keyboard PCB many thousands of times per second. The matrix arrangement of the MEGA65 keyboard does not use the C65 matrix layout.

Instead, the CPLD also sorts the natural matrix of the keyboard into the C65 keyboard matrix order, and transmits this serially via the keyboard cable to the MEGA65 mainboard. The MEGA65 core reads this serial data and uses it to reconstruct a C65-compatible virtual keyboard in the FPGA. This virtual keyboard also takes input from the on-screen-keyboard, synthetic keyboard injection mechanism and/or other keyboard input sources depending on the MEGA65 model.

The end-to-end latency of the keyboard is less than one millisecond.

### C65 KEYBOARD MATRIX

The MEGA65 keyboard presents to legacy software as a C65-compatible keyboard. In this mode all keys are available for standard PETSCII scanning as per normal. There is also a hardware accelerated mechanism for detecting arbitrary combinations of keys that are held down. This is via \$D614 (decimal 54804). Writing a value between 0 and 8 to this register selects the corresponding row of the C65 keyboard matrix, which can then be read back from \$D613. If a bit is zero, then it means that the key is being pressed. If the bit is one, then the key is not being pressed.

The left and up cursor keys are special, because they logically press cursor right or down, and the right shift key. To be able to differentiate between these two situations, you can read \$D60F: Bit 0 is the state of the left cursor key and bit 1 is the state of the up cursor key.

The C65 keyboard matrix layout is as follows:

	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>
<b>0</b>	INST DEL	3	5	7	9	+	£	1	NO SCROLL
<b>1</b>	RETURN	W	R	Y	I	P	*	←	TAB
<b>2</b>	→	A	D	G	J	L	;	CTRL	ALT
<b>3</b>	F7	4	6	8	0	-	CLR HOME	2	HELP
<b>4</b>	F1	Z	C	B	M	.	SHIFT right	SPC	F9
<b>5</b>	F3	S	F	H	K	:	=	◀	F11
<b>6</b>	F5	E	T	O	U	0	@	↑	F13
<b>7</b>	↓	SHIFT left	X	V	N	,	/	RUN STOP	ESC

Note that the keyboard matrix is identical to the C64 keyboard matrix, except for the addition of one extra column on the right hand-side. The cursor left and up keys on the MEGA65 and C65 are implemented as cursor right and down, but with the right shift key applied. This enables them to work from in C64 mode. The **CAPS LOCK** key is not part of the matrix, but has its own dedicated line. Its status can be read from bit 6 of register \$D611 (decimal 54801):

The numbers across the top indicate the columns of the matrix, and the numbers down the left indicate the rows. The unique scan code of a key is calculated by multiplying the column by eight, and adding the row. For example, the **CLR HOME** key is in column 6 and row 3. Thus its scan code is  $6 \times 8 + 3 = 51$ .

## SYNTHETIC KEY EVENTS

The MEGA65 keyboard interface logic allows the use of a variety of keyboard types and alternatives. This is partly to cater for the early development on general purpose FPGA boards, the MEGAphone with its touch interface, and the desktop versions of the MEGA65 architecture. The depressing of up to 3 three keys can be simulated via the registers \$D615 - \$D617 (decimal

54,805 – 54,807). By setting the lower 7 bits of these registers to any C65 keyboard scan code, the MEGA65 will behave as though that key is being held down. To release a key, write \$7F (decimal 127) to the register containing the active key press. For example, to simulate briefly pressing the \* key, the following could be used:

```
POKE DEC("D615"),6*8+1:FOR I=1TO100:NEXT:POKE DEC("D615"),127
```

The FOR loop provides a suitable delay to simulate holding the key for a short time. All statements should be on a single line like this, if entered directly into the BASIC interpreter, because otherwise the MEGA65 will continue to act as though the \* key is being held down, making it rather difficult to enter the other commands!

## KEYBOARD LED CONTROL

The LEDs on the MEGA65's keyboard are normally controlled automatically by the system. However, it is also possible to place them under user control. This is activated by setting bit 7 (decimal 128) of \$D61D (decimal 54813). The lower bits indicate which keyboard LED to set. Values 0 through 11 correspond to the red, green and blue channels of the four LEDs. The table below shows the specific values:

- 0** left-half of DRIVE LED, RED
- 1** left-half of DRIVE LED, GREEN
- 2** left-half of DRIVE LED, BLUE
- 3** right-half of DRIVE LED, RED
- 4** right-half of DRIVE LED, GREEN
- 5** right-half of DRIVE LED, BLUE
- 6** left-half of POWER LED, RED
- 7** left-half of POWER LED, GREEN
- 8** left-half of POWER LED, BLUE
- 9** right-half of POWER LED, RED
- 10** right-half of POWER LED, GREEN
- 11** right-half of POWER LED, BLUE

Register \$D61E (decimal 54814) is used to specify the intensity that should be given to a specific LED (value between 0 and 255).

Note that whatever value is in \$D61E gets written to whatever register is currently selected in \$D61D. Therefore to safely change the intensity of one specific LED ensure \$D61D is set to 255 first. This prevents affecting another LED when we set the intended intensity value into \$D61E. Now select the target LED by setting \$D61D to  $128 + x$ , where  $x$  is a value from the table above. Hold the \$D61D, \$D61E configuration for approximately one millisecond to give the keyboard logic enough time to pick up the new intensity value for the selected LED.

To return the keyboard LEDs to hardware control, clear bit 7 of \$D61D.

For example to pulse the keyboard LEDs red and blue, the following programme could be used:

```
10 REM ENABLE SOFTWARE CONTROL OF LEDS
20 POKEDEC("D61D"),128
30 REM SET ALL LEDS TO OFF
40 POKEDEC("D61E"),0
50 FOR I=0TO11:POKEDEC("D61D"),128+I:NEXT
60 REM SELECT RED CHANNEL OF RIGHT MOST LED
70 POKEDEC("D61D"),128
80 REM CYCLE FROM BLACK TO RED AND BACK
90 FOR I=0TO255:POKEDEC("D61E"),I:NEXT
100 FOR I=255T00STEP-1:POKEDEC("D61E"),I:NEXT
110 REM SELECT BLUE CHANNEL OF LEFT MOST LED
120 POKEDEC("D61D"),128+8
130 REM CYCLE FRO BLACK TO BLUE AND BACK
140 FOR I=0TO255:POKEDEC("D61E"),I:NEXT
150 FOR I=255T00STEP-1:POKEDEC("D61E"),I:NEXT
160 GOT070
```

## NATIVE KEYBOARD MATRIX

The native keyboard matrix is accessible only from the CPLD on the MEGA65's keyboard. If you are programming the MEGA65 computer, you should not need to use this.



0	F5	28	B
1	9	29	← (cursor left)
2	I	30	+
3	K	31	:
4	<	32	NO SCRL
5	INST DEL	33	5
6	CLR HOME	34	R
7	○	35	F
8	F3	36	V
9	8	37	SPACE
10	U	38	O
11	J	39	L
12	M	40	CAPS LOCK
13	→	41	4
14	£	42	E
15	=	43	D
16	F1	44	C
17	7	45	Reserved
18	Y	46	HELP
19	H	47	RETURN
20	N	48	ALT
21	↓	49	3
22	-	50	W
23	;	51	S
24	Reserved	52	X
25	6	53	↑ (cursor up)
26	T	54	F13
27	G	55	↑ (next to *)

**56**  **57** 2  
**58** Q  
**59** A  
**60** Z  
**61** right   
**62**   
**63** \*  
**64** Reserved  
**65** 1  
**66** Reserved  
**67** Reserved

**68** left  and   
**69** /  
**70**   
**71** @  
**72**   
**73**  (next to 1)  
**74**   
**75**   
**76**   
**77** >  
**78**   
**79** P

# E

## APPENDIX

### Decimal, Binary and Hexadecimal

- **Numbers**
- **Notations and Bases**
- **Operations**
- **Signed and Unsigned Numbers**
- **Bit-wise 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 specialised 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 colour 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 colour 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 bit-wise logical operations. See Appendix N Reference Tables as it contains a Base Con-

version 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 sub-scripted 10, binary numbers will have a sub-scripted 2, and hexadecimal numbers will have a sub-scripted 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 N 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 equalling the right digit and the high nibble equalling 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 summarised 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 summarised 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 summarised 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 summarised 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 focused 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 trade-off 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.

# BIT-WISE 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
<b>AND</b> %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.

%	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0
<b>OR</b>	<b>%</b>	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
<hr/>																
%	0	0	0	0	0	0	0	0	1	1	1	0	0	1	1	0

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.

%	0	0	1	1	0	1	1	0	0	0	0	1	0	1	1	1
<b>XOR</b>	<b>%</b>	0	0	1	0	0	0	1	1	0	0	1	1	0	1	0
<hr/>																
%	0	0	0	1	0	1	1	0	1	1	0	0	1	1	0	0

Notice that when the bits are equal the resulting bit is 0. Otherwise the resulting bit is 1.

Much of the utility of these bit-wise 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 bit-wise 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 8,65: 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 = 8 TO 15
100 : P(N) = 2 ↑ N
110 NEXT N
120 REM *** OUTPUT MAIN MENU ***
130 PRINT CHR$(147)
140 PRINT: PRINT "INTEGER BASE CONVERTER"
150 L = 22: GOSUB 1330: PRINT L$
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 M$
440 IF (M$="D") THEN GOSUB 1280: GOTO 380
450 IF (M$="B") THEN GOSUB 1670: GOTO 380
460 IF (M$="H") THEN GOSUB 1800: GOTO 380
470 IF (M$="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: 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 BS
1590 : IF BS=="X" THEN RETURN
1600 : IF BS=" " THEN GOTO 1620
1610 : GOTO 1580
1620 NEXT J
1630 PRINT CHR$(19): FOR I = 1 TO 22: PRINT: NEXT I
1640 PRINT SPC(20); "COMPLETE. PRESS SPACEBAR TO CONTINUE..."
1650 GET BS: IF BS<>" " THEN GOTO 1650: ELSE RETURN
1660 REM *** CONVERT MULTIPLE BINARY INTEGERS ***
1670 IB$=""
1680 PRINT: INPUT "ENTER STARTING BINARY INTEGER (UP TO 16 BITS): ", IB$
1690 IS=IB$: GOSUB 1110: IS"": REM VALIDATE BINARY INPUT
1700 IF (V = 0) THEN GOTO 1680
1710 IB = I
1720 IE$=""
1730 PRINT: INPUT "ENTER ENDING BINARY INTEGER (UP TO 16 BITS): ", IE$
1740 IS=IE$: GOSUB 1110: IS"": REM VALIDATE BINARY INPUT
1750 IF (V = 0) THEN GOTO 1730
1760 IE = I
1770 IF (IE < IB) THEN AS = E$: GOSUB 1810: 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$: GOSUB 1220: HS"": REM VALIDATE HEXADECIMAL INPUT
1880 IF (V = 0) THEN GOTO 1860
1890 HE = H
1900 IF (HE < HB) THEN AS = E$: GOSUB 1810: 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

```

# F

## APPENDIX

# System Memory Map

- **Introduction**
- **MEGA65 Native Memory Map**
- **\$D000 – \$DFFF IO Personalities**
- **CPU Memory Banking**
- **C64/C65 ROM Emulation**



# INTRODUCTION

The MEGA65 computer has a large 28-bit address space, which allows it to address up to 256MiB of memory and memory-mapped devices. This memory map has several different views, depending on which mode the computer is operating in. Broadly, there are five main modes: (1) Hypervisor mode; (2) C64 compatibility mode; (3) C65 compatibility mode; (4) UltiMAX compatibility mode; and (5) MEGA65 mode, or one of the other modes, where the programmer has made use of MEGA65 enhanced features.

It is important to understand that, unlike the C128, the C65 and MEGA65 allow access to all enhanced features from C64 mode, if the programmer wishes to do so. This means that while we frequently talk about "C64 Mode," "C65 Mode" and "MEGA65 Mode," these are simply terms of convenience for the MEGA65 with its memory map (and sometimes other features) configured to provide an environment that matches the appropriate mode. The heart of this is the MEGA65's flexibly memory map.

In this appendix, we will begin by describing the MEGA65's native memory map, that is, where all of the memory, IO devices and other features appear in the 28-bit address space. We will then explain how C64 and C65 compatible memory maps are accessed from this 28-bit address space.

# MEGA65 NATIVE MEMORY MAP

## The First Sixteen 64KB Banks

The MEGA65 uses a similar memory map to that of the C65 for the first MiB of memory, i.e., 16 memory banks of 64KiB each. This is because the C65's 4510 CPU can access only 1MiB of address space. These banks can be accessed from BASIC 10 using the **BANK**, **DMA**, **PEEK** and **POKE** commands. The following table summarises the contents of the first 16 banks:

HEX	DEC	Address	Contents
0	0	\$0xxxx	First 64KiB RAM. This is the RAM visible in C64 mode.
1	1	\$1xxxx	Second 64KiB RAM. This is the 2nd 64KiB of RAM present on a C65.
2	2	\$2xxxx	First half of C65 ROM (C64 mode and shared components) or RAM
3	3	\$3xxxx	Second half of C65 ROM (C65 mode components) or RAM
4	4	\$4xxxx	Additional RAM (384KiB or larger chip-RAM models)
5	5	\$5xxxx	Additional RAM (384KiB or larger chip-RAM models)
6	6	\$6xxxx	Additional RAM (512KiB or larger chip-RAM models)
7	7	\$7xxxx	Additional RAM (512KiB or larger chip-RAM models)
8	8	\$8xxxx	Additional RAM (1MiB or larger chip-RAM models)
9	9	\$9xxxx	Additional RAM (1MiB or larger chip-RAM models)
A	10	\$Axxxx	Additional RAM (1MiB or larger chip-RAM models)
B	11	\$Bxxxx	Additional RAM (1MiB or larger chip-RAM models)
C	12	\$Cxxxx	Additional RAM (1MiB or larger chip-RAM models)
D	13	\$Dxxxx	Additional RAM (1MiB or larger chip-RAM models)

continued ...

...continued

HEX	DEC	Address	Contents
E	14	\$Exxxx	Additional RAM (1MiB or larger chip-RAM models)
F	15	\$Fxxxx	Additional RAM (1MiB or larger chip-RAM models)

The key features of this address space are the 128KiB of RAM in the first two banks, which is also present on the C65. If you intend to write programmes which can also run on a C65, you should only use these two banks of RAM.

On all models it is possible to use all or part of the 128KiB of "ROM" space as RAM. To do this, you must first request that the Hypervisor removes the read-only protection on this area, before you will be able to change its contents. If you are writing a programme which will start from C64 mode, or otherwise switch to using the C64 part of the ROM, instead of the C65 part), then the second half of that space, i.e., BANK 3, can be safely used for your programmes. This gives a total of 192KiB of RAM, which is available on all models of the MEGA65.

On models that have 384KiB or more of chip RAM, BANK 4 and 5 are also available. Similarly, models which provide 1MiB or more of chip RAM will have BANK 6 through 15 also available, giving a total of 896KiB (or 960KiB, if only the C64 part of the ROM is required) of RAM available for your programmes. Note that the MEGA65's built-in freeze cartridge currently freezes only the first 384KiB of RAM.

## Colour RAM

The MEGA65's VIC-IV video controller supports much larger screens than the VIC-II or VIC-III. For this reason, it has access to a separate colour RAM, similar to on the C64. For compatibility with the C65, the first two kilo-bytes of this are accessible at \$1F800 - \$1FFFF. The full 32KiB or 64KiB of colour RAM is located at \$FF80000. This is most easily accessed through the use of advanced DMA operations, or the 32-bit base-page indirect addressing mode of the processor.

At the time of writing, the **BANK** and **DMA** commands cannot be used to access the rest of the colour RAM, because the colour RAM is not located in the first mega-byte of address space. This may be corrected in a future revision of the MEGA65, allowing access to the full colour RAM via BANK 15 or an equivalent DMA job.

# 28-bit Address Space

In addition to the C65-style 1MB address space, the MEGA65 extends this to 256MiB, by using 28-bit addresses. The following shows the high-level layout of this address space.

HEX	DEC	Size	Contents
0000000	0	1	CPU IO Port Data Direction Register
0000001	1	1	CPU IO Port Data
0000002 - 005FFFF	2 - 384 KiB	384 KiB	Fast chip RAM (40MHz)
0060000 - 0FFFFF	384 KiB - 16 MiB	15.6 MiB	Reserved for future chip RAM expansion
1000000 - 3FFFFFF	16 MiB - 64 MiB	48 MiB	Reserved
4000000 - 7FFFFFF	64 MiB - 128 MiB	64 MiB	Cartridge port and other devices on the slow bus (1 - 10 MHz)
8000000 - 87FFFFFF	128 MiB - 135 MiB	8 MiB	8MB ATTIC RAM (selected models only)
8800000 - 8FFFFFF	135 MiB - 144 MiB	8 MiB	8MB CELLAR RAM (selected models only)
9000000 - EFFFFFF	144 MiB - 240 MiB	96 MiB	Reserved for future expansion RAM
F000000 - FF7DFFF	240 MiB - 255.49 MiB	15.49 MiB	Reserved for future IO expansion
FF7E000 - FF7EFFF	255.49 MiB - 255.49 MiB	4 KiB	VIC-IV Character ROM (write only)
FF80000 - FF87FFF	255.5 MiB - 255.53 MiB	32 KiB	VIC-IV Colour RAM (32KiB colour RAM models)
FF88000 - FF8FFFF	255.53 MiB - 255.57 MiB	32 KiB	Additional VIC-IV Colour RAM (64KiB colour RAM models only)
FF90000 - FFCAFFF	255.53 MiB - 255.80 MiB	216 KiB	Reserved
FFCB000 - FFCBFFF	255.80 MiB - 255.80 MiB	4 KiB	Emulated C1541 RAM
FFCC000 - FFCFFFF	255.80 MiB - 255.81 MiB	16 KiB	Emulated C1541 ROM
FFD0000 - FFD0FFF	255.81 MiB - 255.81 MiB	4 KiB	C64 \$Dxxx IO Personality

continued ...

...continued

<b>HEX</b>	<b>DEC</b>	<b>Size</b>	<b>Contents</b>
FFD1000 - FFD1FFF	255.81 MiB - 255.82 MiB	4 KiB	C65 \$Dxxx IO Personality
FFD2000 - FFD2FFF	255.82 MiB - 255.82 MiB	4 KiB	MEGA65 \$Dxxx Ethernet IO Personality
FFD3000 - FFD3FFF	255.82 MiB - 255.82 MiB	4 KiB	MEGA65 \$Dxxx Normal IO Personality
FFD4000 - FFD5FFF	255.82 MiB - 255.83 MiB	8 KiB	Reserved
FFD6000 - FFD67FF	255.83 MiB - 255.83 MiB	2 KiB	Hypervisor scratch space
FFD6000 - FFD6BFF	255.83 MiB - 255.83 MiB	3 KiB	Hypervisor scratch space
FFD6C00 - FFD6DFF	255.83 MiB - 255.83 MiB	512	F011 floppy controller sector buffer
FFD6E00 - FFD6FFF	255.83 MiB - 255.83 MiB	512	SD Card controller sector buffer
FFD7000 - FFD7OFF	255.83 MiB - 255.83 MiB	256	MEGAphone r1 I2C peripherals
FFD7100 - FFD71FF	255.83 MiB - 255.83 MiB	256	MEGA65 r2 I2C peripherals
FFD7200 - FFD72FF	255.83 MiB - 255.83 MiB	256	MEGA65 HDMI I2C registers (only for models fitted with the ADV7511 HDMI driver chip)
FFD7300 - FFD7FFF	255.83 MiB - 255.84 MiB	3.25 KiB	Reserved for future I2C peripherals
FFD8000 - FFDBFFF	255.83 MiB - 255.86 MiB	16 KiB	Hypervisor ROM (only visible in Hypervisor Mode)
FFDC000 - FFDDFFF	255.86 MiB - 255.87 MiB	8 KiB	Reserved for Hypervisor Mode ROM expansion
FFDE000 - FFDE7FF	255.87 MiB - 255.87 MiB	2 KiB	Reserved for Ethernet buffer expansion
FFDE800 - FFDEFFF	255.87 MiB - 255.87 MiB	2 KiB	Ethernet frame read buffer (read only) and Ethernet frame write buffer (write only)
FFDF000 - FFDFFFF	255.87 MiB - 255.87 MiB	4 KiB	Virtual FPGA registers (selected models only)

continued ...

...continued

<b>HEX</b>	<b>DEC</b>	<b>Size</b>	<b>Contents</b>
FFE0000 - FFFFFFFF	255.87 MiB - 256 MiB	128 KiB	Reserved

## \$D000 - \$DFFF IO PERSONALITIES

The MEGA65 supports four different IO personalities. These are selected by writing the appropriate values to the \$D02F KEY register, which is visible in all four IO personalities. Refer to Chapter 9 for more information about the use of the KEY register.

The following table shows which IO devices are visible in each of these IO modes, as well as the KEY register values that are used to select the IO personality.

<b>HEX</b>	<b>C64</b>	<b>C65</b>	<b>MEGA65 ETHERNET</b>	<b>MEGA65</b>
KEY	\$00	\$A5, \$96	\$45, \$54	\$47, \$53
\$D000 - \$D02F	VIC-II	VIC-II	VIC-II	VIC-II
\$D030 - \$D07F	VIC-II <sup>1</sup>	VIC-III	VIC-III	VIC-III
\$D080 - \$D08F	VIC-II	F011	F011	F011
\$D090 - \$D09F	VIC-II	-	SD Card	SD Card
\$D0A0 - \$D0FF	VIC-II	RAM EXPAND CONTROL	-	-
\$D100 - \$D1FF	VIC-II	RED Palette	RED Palette	RED Palette
\$D200 - \$D2FF	VIC-II	GREEN Palette	GREEN Palette	GREEN Palette
\$D300 - \$D3FF	VIC-II	BLUE Palette	BLUE Palette	BLUE Palette
\$D400 - \$D41F	SID Right #1	SID Right #1	SID Right #1	SID Right #1
\$D420 - \$D43F	SID Right #2	SID Right #2	SID Right #2	SID Right #2
\$D440 - \$D45F	SID Left #1	SID Left #1	SID Left #1	SID Left #1
\$D460 - \$D47F	SID Left #2	SID Left #2	SID Left #2	SID Left #2
\$D480 - \$D49F	SID Right #1	SID Right #1	SID Right #1	SID Right #1
\$D4A0 - \$D4BF	SID Right #2	SID Right #2	SID Right #2	SID Right #2
\$D4C0 - \$D4DF	SID Left #1	SID Left #1	SID Left #1	SID Left #1
\$D4E0 - \$D4FF	SID Left #2	SID Left #2	SID Left #2	SID Left #2
\$D500 - \$D5FF	SID images	-	Reserved	Reserved
\$D600 - \$D63F	-	UART	UART	UART
\$D640 - \$D67F	-	UART images	HyperTrap Registers	HyperTrap Registers
\$D680 - \$D6FF	-	-	MEGA65 Devices	MEGA65 Devices
\$D700 - \$D7FF	-	-	MEGA65 Devices	MEGA65 Devices
\$D800 - \$DBFF	COLOUR RAM	COLOUR RAM	ETHERNET Buffer	COLOUR RAM
\$DC00 - \$DDFF	CIAs	CIAs / COLOUR RAM	ETHERNET Buffer	CIAs / COLOUR RAM
\$DE00 - \$DFFF	CART IO	CART IO	ETHERNET Buffer	CART IO / SD SECTOR

<sup>1</sup> In the C64 IO personality, \$D030 behaves as on C128, allowing toggling between 1MHz and 2MHz CPU speed.

<sup>2</sup> The additional MEGA65 SIDs are visible in all IO personalities.

<sup>3</sup> Some models may replace the repeated images of the first four SIDs with four additional SIDs, for a total of 8 SIDs.

# CPU MEMORY BANKING

The 45GS10 processor, like the 6502, can only “see” 64KB of memory at a time. Access to additional memory is via a selection of bank-switching mechanisms. For backward-compatibility with the C64 and C65, the memory banking mechanisms for both of these computers existing the MEGA65:

1. C65-style MAP instruction banking
2. C65-style \$D030 banking
3. C64-style cartridge banking
4. C64-style \$00 / \$01 banking

It is important to understand that these different banking modes have a priority order: If a higher priority form of banking is being used, it takes priority over a lower priority form. The C65 banking methods take priority of the C64 mode banking methods. So, for example, if the 45GS10 MAP instruction has been used to provide a particular memory layout, the C64-style \$00 / \$01 banking will not be visible.

This makes the overall banking scheme more complex than on the C64. Thus to understand what the actual memory layout will be, you should start by considering the effects of C64 memory banking, and then if any C65 MAP instruction memory banking is enabled, using that to override the C64-style memory banking. Then if any C65 \$D030 memory banking is used, that overrides both the C64 and C65 MAP instruction memory banking. Finally, if IO is banked, or if there are any cartridges inserted and active, their effects are made.

The following diagram shows the different types of banking that can apply to the different areas of the 64KB that the CPU can see. The higher layers take priority over the lower layers, as described in the previous paragraph.

IO/CART		CART ROMLO	CART ROMHI			IO	CART ROMHI
C65		BASIC	BASIC	INTER-FACE			KERNAL
MAP	MAP LO (4 x 8KB slabs)	MAP HI (4 x 8KB slabs)					
C64			BASIC		CHAR ROM	KERNAL	
RAM	RAM*	RAM	RAM	RAM	RAM	RAM	RAM
	\$0000 - \$7FFF	\$8000 - \$9FFF	\$A000 - \$BFFF	\$C000 - \$CFFF	\$D000 - \$DFFF	\$E000 - \$FFFF	

(There are actually a few further complications. For example, if the cartridge selects the UltiMAX(tm) game mode, then only the first 4KB of RAM will be visible, and the remaining address space will be un-mapped, and able to be supplied by the cartridge.)

For example, using \$D030 to bank in C65 ROM at \$A000, this will take priority over the C64 BASIC 2 ROM at the same address.

## C64/C65 ROM EMULATION

The C64 and C65 use ROM memories to hold the KERNAL and BASIC system. The MEGA65 is different: It uses 128KB of its 384KB fast chip RAM at \$20000 - \$3FFFF (banks 2 and 3) to hold these system programmes. This makes it possible to change or upgrade the "ROM" that the MEGA65 is running, without having to open the computer. It is even possible to use the MEGA65's Freeze Menu to change the "ROM" being used while a programme is running.

The C64 and C65 memory banking methods use this 128KB of area when making ROM banks visible. When the RAM banks are mapped, they are always read-only. However, if the MAP instruction or DMA is used to access that address area, it is possible to write to it. For improved backward compatibility, the whole 128KB region of memory is normally set to read-only.

A programme can, however, request read-write access to this 128KB area of memory, so that it can make full use of the MEGA65's 384KB of chip RAM. This is accomplished by triggering the *Toggle Rom Write-protect* system trap of the hypervisor. The following code-fragment demonstrates how to do this. Calling it a second time will re-activate the write-protection.

```
LDA #$70  
STA $D640  
MOP
```

This fragment works by calling sub-function \$70 (toggle ROM write-protect) of Hypervisor trap \$00. Note that the **MOP** is mandatory. The MEGA65 IO personality must be first selected, so that the \$D640 register is un-hidden.

The current write-protection state can be tested by attempting to write to this area of memory. Also, you can examine and toggle the current state from in the MEGA65 Freeze Menu.

# C65 Compatibility ROM Layout

The layout of the C65 compatibility 128KB ROM area is identical to that of the C65:

HEX	Contents
\$3E000 - \$3FFFF	C65 KERNEL
\$3C000 - \$3DFFF	RESERVED
\$38000 - \$3BFFF	C65 BASIC GRAPHICS ROUTINES
\$32000 - \$37FFF	C65 BASIC
\$30000 - \$31FFF	MONITOR (gets mapped at \$6000 - \$7FFF)
\$2E000 - \$2FFFF	C64 KERNEL
\$2D000 - \$2DFFF	C64 CHARSET
\$2C000 - \$2CFFF	INTERFACE
\$24000 - \$27FFF	RESERVED
\$20000 - \$23FFF	DOS (gets mapped at \$8000 - \$BFFF)

The INTERFACE program is a series of routines that are used by the C65 to switch between C64 mode, C65 mode and the C65's built-in DOS. The DOS is located in the lower-eighth of the ROM.



## APPENDIX

# 45GS02 Microprocessor

- **Introduction**
- **Differences to the 6502**
- **C64 CPU Memory Mapped Registers**
- **New CPU Memory Mapped Registers**
- **MEGA65 CPU Maths 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 45GS02 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 Hyppo 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 45GS02 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 up to 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 I

# 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

<b>\$D030 Bit</b>	<b>Signal Name</b>	<b>20-bit Address</b>	<b>16-bit Address</b>	<b>Read-Write Access?</b>
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 J.

### **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 I

### **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.

XXX - Diagram needed for this.

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 = \$D8000. 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
LDX #$0f
LDY #$00
LDZ #$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 = $08000
LDA #$80
LDX #$8d
LDY #$00
LDZ #$00
Map
EOM

; Ethernet buffer now visible at $6800 - $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 preceded 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

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
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				-		CARTEN	BRCOST	
D7FD	55293	NOEXROM	POWEREN				-		

- **BRCOST** 1=charge extra cycle(s) for branches taken
- **CARTEN** 1= enable cartridges
- **HTRAPO0** Writing triggers hypervisor trap \$00
- **HTRAPO1** Writing triggers hypervisor trap \$01
- **HTRAPO2** Writing triggers hypervisor trap \$02
- **HTRAPO3** Writing triggers hypervisor trap \$03
- **HTRAPO4** Writing triggers hypervisor trap \$04
- **HTRAPO5** Writing triggers hypervisor trap \$05
- **HTRAPO6** Writing triggers hypervisor trap \$06
- **HTRAPO7** Writing triggers hypervisor trap \$07
- **HTRAPO8** Writing triggers hypervisor trap \$08
- **HTRAPO9** Writing triggers hypervisor trap \$09
- **HTRAPOA** Writing triggers hypervisor trap \$0A
- **HTRAPOB** Writing triggers hypervisor trap \$0B
- **HTRAPOC** Writing triggers hypervisor trap \$0C
- **HTRAPOD** Writing triggers hypervisor trap \$0D
- **HTRAPOE** Writing triggers hypervisor trap \$0E
- **HTRAPOF** 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
- **POWEREN** Set to zero to power off computer on supported systems.  
WRITE ONLY.
- **SPEEDBIAS** 1/2/3.5MHz CPU speed fine adjustment

## MEGA65 CPU MATHS UNIT REGISTERS

Some models include a math unit, which helps to accelerate the calculation of fixed-point formulae. This is not present in all models of the MEGA65.

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
D770	55152								MULTINA
D771	55153								MULTINA

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D772	55154					MULTINA			
D773	55155					-			MULTINA
D774	55156					MULTINB			
D775	55157					MULTINB			
D776	55158				-			MULTINB	
D778	55160					MULTOUT			
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			

continued ...

...continued

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
D79A	55194					MATHIN6			
D79B	55195					MATHIN6			
D79C	55196					MATHIN7			
D79D	55197					MATHIN7			
D79E	55198					MATHIN7			
D79F	55199					MATHIN7			
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			

continued ...

...continued

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
D7C0	55232		UNIT0INB					UNIT0INA	
D7C1	55233		UNIT1INB					UNIT1INA	
D7C2	55234		UNIT2INB					UNIT2INA	
D7C3	55235		UNIT3INB					UNIT3INA	
D7C4	55236		UNIT4INB					UNIT4INA	
D7C5	55237		UNIT5INB					UNIT5INA	
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 B (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 con-

tents 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 rivalling – 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

continued ...

...continued

<b>HEX</b>	<b>DEC</b>	<b>Name</b>	<b>Description</b>
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	SECUREEXIT	Leave secure container trap entry point.
804C	32844	SYSCALL13	SYSCALL 19 entry point
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

continued ...

...continued

<b>HEX</b>	<b>DEC</b>	<b>Name</b>	<b>Description</b>
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
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 break-point 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 will typically use the last 512 bytes for the stack and zero-page, giving an overall memory map as follows:

<b>HEX</b>	<b>DEC</b>	<b>Description</b>
8000 - 81FF	32768 - 33279	SYSCALL and trap entry points
8200 - BDFF	33280 - 48639	Available for hypervisor or operating system program

continued ...

...continued

<b>HEX</b>	<b>DEC</b>	<b>Description</b>
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

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
D640	54848								REGA
D641	54849								REGX
D643	54851								REGZ
D644	54852								REGB
D645	54853								SPL
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								DMADSTMB
D655	54869								DMALADDR

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
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			
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
- **SPL** Hypervisor SPL 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() {
```

```
    // Initialise 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 message 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 XMeg65 emulator with a command like:

```
xmeg65 -kickup mega65hyper.bin
```



# H

## APPENDIX

# 45GS02 & 6502 Instruction Sets

- Addressing Modes
- 45GS02 Instruction Set
- 6502 Instruction Set



The 45GS02 CPU is able to operate in native mode, where it is compatible with the CSG 4510, and in 6502 compatibility mode, where 6502 undocumented instructions, also known as illegal instructions, are supported for compatibility. The instruction set and timing information for both personalities are listed in the following sections.

## ADDRESSING MODES

The 45GS02 supports 18 different addressing modes, which are explained below.

### Implied

In this mode, there are no operands, as the precise function of the instruction is implied by the instruction itself. For example, the `INX` instruction increments the X Register.

### Accumulator

In this mode, the Accumulator is the operand. This is typically used to shift, rotate or modify the value of the Accumulator Register in some way. For example, `INC A` increments the value in the Accumulator Register.

### Immediate Mode

In this mode, the argument to the instruction is a value that is used directly. This is indicated by proceeding the value with a # character. Most assemblers allow values to be entered in decimal, or in hexadecimal by preceding the value with a \$ sign, in binary, by preceding the value with a % sign. For example, to set the Accumulator Register to the value 5, you could use the following:

```
LDA #5
```

The immediate argument is encoded as a single byte following the instruction. For the above case, the instruction stream would contain \$A9, the opcode for LDA immediate mode, followed by \$05, the immediate operand.

## Immediate Word Mode

In this mode, the argument is a 16-bit value that is used directly. There is only one instruction which uses this addressing mode, `PHW`. For example, to push the word \$1234 onto the stack, you could use:

```
PHW #$1234
```

The low-byte of the immediate value follows the opcode of the instruction. The high-byte of the immediate value then follows that. For the above example, the instruction stream would thus be \$F4 \$34 \$12.

## Base Page (Zero-Page) Mode

In this mode, the argument is an 8-bit address. The upper 8-bits of the address are taken from the Base Page Register. On 6502 processors, there is no Base Page Register, and instead, the upper 8-bits are always set to zero - hence the name of this mode on the 6502: Zero-Page. On the 45GS02, it is possible to move this "Zero-Page" to any page in the processor's 64KB view of memory by setting the Base Page Register using the `TAB` instruction. Base Page Mode allows faster access to a 256 region of memory, and uses less instruction bytes to do so.

The argument is encoded as a single byte that immediately follows the instruction opcode. For example, `LDA $12` would read the value stored in location \$12 in the Base Page, and put it into the Accumulator Register. The instruction byte stream for this would be \$85 \$12.

## Base Page (Zero-Page) X Indexed Mode

This mode is identical to Base Page Mode, except that the address is formed by taking the argument, and adding the value of the X Register to it. In 6502 mode, the result will always be in the Base Page, that is, any carry due to the addition from the low-byte into the high-byte of the address will be ignored. The encoding for this addressing mode is identical to Base Page Mode.

## **Base Page (Zero-Page) Y Indexed Mode**

This mode is identical to Base Page Mode, except that the address is formed by taking the argument, and adding the value of the Y Register to it. In 6502 mode, the result will always be in the Base Page, that is, any carry due to the addition from the low-byte into the high-byte of the address will be ignored. The encoding for this addressing mode is identical to Base Page Mode.

## **Base Page (Zero-Page) Z Indexed Mode**

This mode is identical to Base Page Mode, except that the address is formed by taking the argument, and adding the value of the Z Register to it. In 6502 mode, the result will always be in the Base Page, that is, any carry due to the addition from the low-byte into the high-byte of the address will be ignored. The encoding for this addressing mode is identical to Base Page Mode.

## **Absolute Mode**

In this mode, the argument is an 16-bit address. The low 8-bits of the address are taken from the byte immediately following the instruction opcode. The upper 8-bits are taken from the byte following that. For example, the instruction LDA \$1234, would read the memory location \$1234, and place the read value into the Accumulator Register. This would be encoded as \$AD \$34 \$12.

## **Absolute X Indexed Mode**

This mode is identical to Absolute Mode, except that the address is formed by taking the argument, and adding the value of the X Register to it. If the indexing causes the address to cross a page boundary, i.e., if the upper byte of the address changes, this may incur a 1 cycle penalty, depending on the processor mode and speed setting. The encoding for this addressing mode is identical to Absolute Mode.

## Absolute Y Indexed Mode

This mode is identical to Absolute Mode, except that the address is formed by taking the argument, and adding the value of the Y Register to it. If the indexing causes the address to cross a page boundary, i.e., if the upper byte of the address changes, this may incur a 1 cycle penalty, depending on the processor mode and speed setting. The encoding for this addressing mode is identical to Absolute Mode.

## Absolute Z Indexed Mode

This mode is identical to Absolute Mode, except that the address is formed by taking the argument, and adding the value of the Z Register to it. If the indexing causes the address to cross a page boundary, i.e., if the upper byte of the address changes, this may incur a 1 cycle penalty, depending on the processor mode and speed setting. The encoding for this addressing mode is identical to Absolute Mode.

## Absolute Indirect Mode

In this mode, the 16-bit argument is the address that points to, i.e., contains the address of actual byte to read. For example, if memory location \$1234 contains \$78 and memory location \$1235 contains \$56, then `JMP ($1234)` would jump to address \$5678. The encoding for this addressing mode is identical to Absolute Mode.

## Absolute Indirect X-Indexed Mode

In this mode, the 16-bit argument is the address that points to, i.e., contains the address of actual byte to read. It is identical to Absolute Indirect Mode, except that the value of the X Register is added to the pointer address. For example, if the X Register contains the value \$04, memory location \$1238 contains \$78 and memory location \$1239 contains \$56, then `JNP ($1234)` would jump to address \$5678. The encoding for this addressing mode is identical to Absolute Mode.

## **Base Page Indirect X-Indexed Mode**

This addressing mode is identical to Absolute Indirect X-Indexed Mode, except that the address of the pointer is formed from the Base Page Register (high byte) and the 8-bit operand (low byte). The encoding for this addressing mode is identical to Base Page Mode.

## **Base Page Indirect Y-Indexed Mode**

This addressing mode differs from the X-Indexed Indirect modes, in that the Y Register is added to the address that is read from the pointer, instead of being added to the pointer. This is a very useful mode, that is frequently because it effectively provides access to “the Y-th byte of the memory at the address pointed to by the operand.” That is, it de-references a pointer. The encoding for this addressing mode is identical to Base Page Mode.

## **Base Page Indirect Z-Indexed Mode**

This addressing mode differs from the X-Indexed Indirect modes, in that the Z Register is added to the address that is read from the pointer, instead of being added to the pointer. This is a very useful mode, that is frequently because it effectively provides access to “the Z-th byte of the memory at the address pointed to by the operand.” That is, it de-references a pointer. The encoding for this addressing mode is identical to Base Page Mode.

That is, it is equivalent to the Base Page Indirect Y-Indexed Mode.

## **32-bit Base Page Indirect Z-Indexed Mode**

This mode is formed by preceding a Base Page Indirect Z-Indexed Mode instruction with the NOP instruction (opcode \$EA). This causes the 45GS02 to

read a 32-bit address instead of a 16-bit address from the Base Page address indicated by its operand. The Z index is added to that pointer. Importantly, the 32-bit address does not refer to the processor's current 64KB view of memory, but rather to the 45GS02's true 28-bit address space. This allows easy access to any memory, without requiring the use of complex bank-switching or DMA operations.

For example, if addresses \$12 to \$15 contained the bytes \$20, \$D0, \$FF, \$0D, and the Z index contained the value \$01, the following instruction sequence would change the screen colour to blue:

```
LDA #$06  
NOP  
STA ($12),Z
```

The encoding for this addressing mode is identical to Base Page Mode.

## Stack Relative Indirect, Indexed by Y

This addressing mode is similar to Base Page Indirect Y-Indexed Mode, except that instead of providing the address of the pointer in the Base Page, the operand indicates the offset in the stack to find the pointer. This addressing mode effectively de-references a pointer that has been placed on the stack, e.g., as part of a function call from a high-level language. It is encoded identically to the Base Page Mode.

## Relative Addressing Mode

In this addressing mode, the operand is an 8-bit signed offset to the current value of the Programme Counter (PC). It is used to allow branches to encode the near-by address at which execution should proceed if the branch is taken.

## Relative Word Addressing Mode

This addressing mode is identical to Relative Addressing Mode, except that the address offset is a 16-bit value. This allows a relative branch or jump to any location in the current 64KB memory view. This makes it possible to write

software that is fully relocatable, by avoiding the need for absolute addresses when calling routines.

# Opcode Map

	\$x0	\$x1	\$x2	\$x3	\$x4	\$x5	\$x6	\$x7	\$x8	\$x9	\$xA	\$xB	\$xC	\$xD	\$xE	\$xF	
	BRK	ORA	CLE	SEE	TSB	ORA	ASL	RMB0	PHP	ORA	ASL	TSY	TSB	ORA	ASL	BRO	
\$0x																	
\$1x	BPL	ORA	ORA	BPL	TRB	ORA	ASL	RMB1	CLC	ORA	INC	INZ	TRB	ORA	ASL	BR1	
\$2x	JSR	AND	JSR	JSR	BIT	AND	ROL	RMB2	PLP	AND	ROL	TY\$	BIT	AND	ROL	BR2	
\$3x	BMI	AND	BMI	BIT	AND	ROL	RMB3	SEC	AND	DEC	DEZ	BIT	AND	ROL	ROL	BR3	
\$4x	RTI	EOR	NEG	ASR	ASR	EOR	LSR	RMB4	PHA	EOR	LSR	TAZ	JMP	EOR	LSR	BR4	
\$5x	BVC	EOR	BVC	ASR	EOR	LSR	RMB5	CLI	EOR	PHY	TAB	MAP	EOR	LSR	BR5		
\$6x	RTS	ADC	RTS	BSR	STZ	ADC	ROR	RMB6	PLA	ADC	ROR	TZA	JMP	ADC	ROR	BR6	
\$7x	BVS	ADC	ADC	BVS	STZ	ADC	ROR	RMB7	SEI	ADC	PLY	TBA	JMP	ADC	ROR	BR7	
\$8x	BRA	STA	STA	BRA	STY	STA	STX	SMB0	DEY	BIT	TXA	STY	STA	STX	BS0		
\$9x	BCC	STA	STA	BCC	STY	STA	STX	SMB1	TYA	STA	TXS	STX	STA	STZ	BBS1		
\$Ax	LDY	LDA	LDX	LDY	LDA	LDX	SMB2	TAY	LDA	TAX	LDZ	LDY	LDA	LDX	BS2		
\$Bx	BCS	LDA	BCS	LDY	LDA	LDX	SMB3	CLV	LDA	TSX	LDZ	LDY	LDA	LDX	BS3		
\$Cx	CPY	CMP	CPZ	DEW	CPY	CMP	DEC	SMB4	INY	CMP	DEX	ASW	CPY	CMP	DEC	BBS4	
\$Dx	BNE	CMP	CMP	BNE	CPZ	CMP	DEC	SMB5	CLD	CMP	PHX	PHZ	CPZ	CMP	DEC	BS5	
\$Ex	CPX	SBC	LDA	INW	CPX	SBC	INC	SMB6	INX	SBC	EOM	ROW	CPX	SBC	INC	BBS6	
\$Fx	BEQ	SBC	SBC	BEQ	PHW	SBC	INC	SMB7	SED	SBC	PLX	PLZ	PHW	SBC	INC	BBS7	

# Instruction Timing

The following table lists the base cycle count for each opcode. Note that the number of cycles depends on the speed setting of the processor: Some instructions take more or fewer cycles when the processor is running at full-speed, or a C65 compatibility 3.5MHz speed, or at C64 compatibility 1MHz/2MHz speed. More detailed information on this is listed under each instruction's information, but the high-level view is:

- When the processor is running at 1MHz, all instructions take at least two cycles, and dummy cycles are re-inserted into Read-Modify-Write instructions, so that all instructions take exactly the same number of cycles as on a 6502.
- The Read-Modify-Write instructions and all instructions that read a value from memory all require an extra cycle when operating at full speed, to allow signals to propagate within the processor.
- The Read-Modify-Write instructions require an additional cycle if the operand is \$D019, as the dummy write is performed in this case. This is to improve compatibility with C64 software that frequently uses this "bug" of the 6502 to more rapidly acknowledge VIC-II interrupts.
- Page-crossing and branch-taking penalties do not apply when the processor is running at full speed.
- Many instructions require fewer cycles when the processor is running at full speed, as generally most non-bus cycles are removed. For example, Pushing and Pulling values to and from the stack requires only 2 cycles, instead of the 4 that the 6502 requires for these instructions.

Note that it is possible that further changes to processor timing will occur.

Similar issues apply to when the processor is in 6502 mode.

	\$x0	\$x1	\$x2	\$x3	\$x4	\$x5	\$x6	\$x7	\$x8	\$x9	\$xA	\$xB	\$xC	\$xD	\$xE	\$xF
\$0x	7	5 <sup>pr</sup>	1 <sup>s</sup>	1 <sup>s</sup>	4 <sup>mrr</sup>	3 <sup>r</sup>	4 <sup>dmr</sup>	4 <sup>r</sup>	3 <sup>m</sup>	2	1 <sup>s</sup>	1 <sup>s</sup>	5 <sup>mrr</sup>	4 <sup>r</sup>	5 <sup>dmr</sup>	4 <sup>br</sup>
\$1x	2 <sup>b</sup>	5 <sup>pr</sup>	5 <sup>pr</sup>	3 <sup>b</sup>	4 <sup>mrr</sup>	3 <sup>pr</sup>	4 <sup>dmpri</sup>	4 <sup>r</sup>	1 <sup>s</sup>	4 <sup>pr</sup>	1 <sup>s</sup>	1 <sup>s</sup>	5 <sup>mrr</sup>	4 <sup>pr</sup>	5 <sup>dmpri</sup>	4 <sup>br</sup>
\$2x	5	5 <sup>r</sup>	7	7	3 <sup>r</sup>	3 <sup>r</sup>	5 <sup>dmr</sup>	4 <sup>r</sup>	3 <sup>m</sup>	2	1 <sup>s</sup>	1 <sup>s</sup>	4 <sup>r</sup>	4 <sup>r</sup>	5 <sup>dmr</sup>	4 <sup>br</sup>
\$3x	2 <sup>b</sup>	5 <sup>pr</sup>	5 <sup>pr</sup>	3 <sup>b</sup>	3 <sup>pr</sup>	3 <sup>pr</sup>	5 <sup>dmpri</sup>	4 <sup>r</sup>	1 <sup>s</sup>	4 <sup>pr</sup>	1 <sup>s</sup>	1 <sup>s</sup>	4 <sup>pr</sup>	4 <sup>pr</sup>	5 <sup>dmpri</sup>	4 <sup>br</sup>
\$4x	5	5 <sup>pr</sup>	2 <sup>m</sup>	2	4 <sup>dmt</sup>	3 <sup>r</sup>	5 <sup>dmr</sup>	4 <sup>r</sup>	3 <sup>m</sup>	2	1 <sup>s</sup>	1 <sup>s</sup>	3	4 <sup>r</sup>	5 <sup>dmr</sup>	4 <sup>br</sup>
\$5x	2 <sup>b</sup>	5 <sup>pr</sup>	5 <sup>pr</sup>	3 <sup>b</sup>	4 <sup>dmpri</sup>	3 <sup>pr</sup>	5 <sup>dmpri</sup>	4 <sup>r</sup>	1 <sup>s</sup>	4 <sup>pr</sup>	3 <sup>m</sup>	1 <sup>s</sup>	1 <sup>s</sup>	4 <sup>pr</sup>	6 <sup>dmpri</sup>	4 <sup>br</sup>
\$6x	4	5 <sup>r</sup>	7 <sup>d</sup>	3 <sup>b</sup>	3	3 <sup>r</sup>	5 <sup>dmpri</sup>	4 <sup>r</sup>	3 <sup>m</sup>	2	1 <sup>s</sup>	1 <sup>s</sup>	5 <sup>m</sup>	4 <sup>r</sup>	5 <sup>dmpri</sup>	4 <sup>br</sup>
\$7x	2 <sup>b</sup>	5 <sup>pr</sup>	5 <sup>pr</sup>	3 <sup>b</sup>	3 <sup>pr</sup>	3 <sup>pr</sup>	5 <sup>dmpri</sup>	4 <sup>r</sup>	1 <sup>s</sup>	5 <sup>pr</sup>	3 <sup>m</sup>	1 <sup>s</sup>	6 <sup>mp</sup>	5 <sup>pr</sup>	5 <sup>dmpri</sup>	4 <sup>br</sup>
\$8x	2 <sup>b</sup>	5 <sup>p</sup>	6 <sup>p</sup>	3 <sup>b</sup>	3	3	4 <sup>r</sup>	1 <sup>s</sup>	2	1 <sup>s</sup>	4 <sup>p</sup>	4	4	4	4	4 <sup>br</sup>
\$9x	2 <sup>b</sup>	5 <sup>p</sup>	5 <sup>p</sup>	3 <sup>b</sup>	3 <sup>p</sup>	3 <sup>p</sup>	3 <sup>p</sup>	4 <sup>r</sup>	1 <sup>s</sup>	4 <sup>p</sup>	1 <sup>s</sup>	4 <sup>p</sup>	4	4 <sup>p</sup>	4 <sup>p</sup>	4 <sup>br</sup>
\$Ax	2	5 <sup>pr</sup>	2	2	3 <sup>r</sup>	3 <sup>r</sup>	3 <sup>r</sup>	4 <sup>r</sup>	1 <sup>s</sup>	2	1 <sup>s</sup>	4 <sup>r</sup>	4 <sup>r</sup>	4 <sup>r</sup>	4 <sup>r</sup>	4 <sup>br</sup>
\$Bx	2 <sup>b</sup>	5 <sup>pr</sup>	5 <sup>pr</sup>	3 <sup>b</sup>	3 <sup>pr</sup>	3 <sup>pr</sup>	5 <sup>pr</sup>	4 <sup>r</sup>	4 <sup>pr</sup>	1 <sup>s</sup>	4 <sup>pr</sup>	4 <sup>pr</sup>	4 <sup>pr</sup>	4 <sup>pr</sup>	4 <sup>br</sup>	
\$Cx	2	5 <sup>pr</sup>	2	7 <sup>dmr</sup>	3 <sup>r</sup>	3 <sup>r</sup>	5 <sup>dmr</sup>	4 <sup>r</sup>	1 <sup>s</sup>	2	1 <sup>s</sup>	7 <sup>dmr</sup>	4 <sup>r</sup>	4 <sup>r</sup>	6 <sup>dmr</sup>	4 <sup>br</sup>
\$Dx	2 <sup>b</sup>	5 <sup>pr</sup>	5 <sup>pr</sup>	3 <sup>b</sup>	3 <sup>r</sup>	3 <sup>pr</sup>	5 <sup>dmpri</sup>	4 <sup>r</sup>	4 <sup>pr</sup>	3 <sup>m</sup>	3 <sup>m</sup>	4 <sup>r</sup>	4 <sup>pr</sup>	6 <sup>dmpri</sup>	4 <sup>br</sup>	
\$Ex	2	3 <sup>mp</sup>	6 <sup>dmpri</sup>	7 <sup>dmr</sup>	3 <sup>r</sup>	3 <sup>r</sup>	5 <sup>dmr</sup>	4 <sup>r</sup>	1 <sup>s</sup>	2	1 <sup>s</sup>	5 <sup>dmr</sup>	4 <sup>r</sup>	4 <sup>r</sup>	6 <sup>dmr</sup>	4 <sup>br</sup>
\$Fx	2 <sup>b</sup>	3 <sup>pr</sup>	3 <sup>pr</sup>	3 <sup>b</sup>	5 <sup>m</sup>	3 <sup>pr</sup>	5 <sup>dmpri</sup>	4 <sup>r</sup>	1 <sup>s</sup>	4 <sup>pr</sup>	3 <sup>m</sup>	3 <sup>m</sup>	7 <sup>m</sup>	4 <sup>pr</sup>	6 <sup>dpr</sup>	4 <sup>br</sup>

*b* Add one cycle if branch crosses a page boundary.

*d* Subtract one cycle when CPU is at 3.5MHz.

*m* Subtract non-bus cycles when at 40MHz.

*p* Add one cycle if indexing crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

# Addressing Mode Table

	\$x0	\$x1	\$x2	\$x3	\$x4	\$x5	\$x6	\$x7	\$x8	\$x9	\$xA	\$xB	\$xC	\$xD	\$xE	\$xF
\$0x																
\$1x	\$rr	(\$mn,X)	(\$mn),Y	(\$mn),Z	\$rr	\$mn	\$mn,X	\$mn	\$mn	#\$mn	A	\$mn	\$mn	\$mn	\$mn	\$mn,\$rr
\$2x	\$mnn	(\$mn,X)	(\$mn)	(\$mn,X)	\$mn	\$mn	\$mn	\$mn	\$mn	\$mn	A	\$mn	\$mn	\$mn	\$mn	\$mn,\$rr
\$3x	\$rr	(\$mn),Y	(\$mn),Z		\$rr	\$mn,X	\$mn,X	\$mn,X	\$mn,X	\$mn	\$mn,Y	A	\$mn	\$mn	\$mn	\$mn,\$rr
\$4x		(\$mn,X)	A			\$mn	\$mn	\$mn	\$mn	\$mn	\$mn,Y	A	\$mn	\$mn	\$mn	\$mn,\$rr
\$5x	\$rr	(\$mn),Y	(\$mn),Z		\$rr	\$mn,X	\$mn,X	\$mn,X	\$mn,X	\$mn	\$mn,Y	A	\$mn	\$mn	\$mn	\$mn,\$rr
\$6x		(\$mn,X)	#\$mn			\$mn	\$mn	\$mn	\$mn	\$mn	#\$mn	A				\$mn,\$rr
\$7x	\$rr	(\$mn),Y	(\$mn),Z		\$rr	\$mn,X	\$mn,X	\$mn,X	\$mn,X	\$mn	\$mn,Y					
\$8x	\$rr	(\$mn,X)	(\$mn,SP),Y		\$rr	\$mn	\$mn	\$mn	\$mn	\$mn	#\$mn		\$mn,X	\$mn	\$mn	\$mn,\$rr
\$9x	\$rr	(\$mn),Y	(\$mn),Z		\$rr	\$mn,X	\$mn,X	\$mn,Y	\$mn	\$mn	\$mn,Y		\$mn	\$mn	\$mn	\$mn,\$rr
\$Ax	#\$mn	(\$mn,X)	#\$mn		#\$mn	\$mn	\$mn	\$mn	\$mn	#\$mn		\$mn	\$mn	\$mn	\$mn	\$mn,\$rr
\$Bx	\$rr	(\$mn),Y	(\$mn),Z		\$rr	\$mn,X	\$mn,X	\$mn,Y	\$mn	\$mn	\$mn,Y		\$mn,X	\$mn	\$mn	\$mn,\$rr
\$Cx	#\$mn	(\$mn,X)	#\$mn	\$mn	\$mn	\$mn	\$mn	\$mn	\$mn	#\$mn	\$mn	\$mn	\$mn	\$mn	\$mn	\$mn,\$rr
\$Dx	\$rr	(\$mn),Y	(\$mn),Z		\$rr	\$mn	\$mn,X	\$mn,X	\$mn	\$mn,Y		\$mn	\$mn,X	\$mn	\$mn	\$mn,\$rr
\$Ex	#\$mn	(\$mn,X)	(\$mn,SP),Y	\$mn	\$mn	\$mn	\$mn	\$mn	#\$mn	\$mn	\$mn	\$mn	\$mn	\$mn	\$mn	\$mn,\$rr
\$Fx	\$rr	(\$mn),Y	(\$mn),Z		\$rr	#\$mn	\$mn,X	\$mn,X	\$mn	\$mn	\$mn,Y		\$mn	\$mn	\$mn	\$mn,\$rr

# ADC

This instruction adds the argument to the contents of the Accumulator Register and the Carry Flag. If the D flag is set, then the addition is performed using Binary Coded Decimal.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, else it will be cleared. If the flag is set, this indicates that a signed overflow has occurred.
- The C flag will be set if the unsigned result is >255, or >99 if the D flag is set.

ADC : Add with carry		4510						
A $\leftarrow$ A+M+C		N	Z	I	C	D	V	E
		+	+	.	+	.	+	.
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>				
(indirect,X)	ADC (\$nn,X)	61	2	5		r		
base-page	ADC \$nn	65	2	3		r		
immediate	ADC #\$nn	69	2	2				
absolute	ADC \$nnnn	6D	3	4		r		
(indirect),Y	ADC (\$nn),Y	71	2	5		pr		
(indirect),Z	ADC (\$nn),Z	72	2	5		pr		
base-page,X	ADC \$nn,X	75	2	3		pr		
absolute,Y	ADC \$nnnn,Y	79	3	5		pr		
absolute,X	ADC \$nnnn,X	7D	3	5		pr		

p Add one cycle if indexing crosses a page boundary.

r Add one cycle if clock speed is at 40 MHz.

# AND

This instruction performs a binary AND operation of the argument with the accumulator, and stores the result in the accumulator. Only bits that were already set in the accumulator, and that are set in the argument will be set in the accumulator on completion.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

AND : Binary AND		4510						
$A \leftarrow A AND M$		N	Z	I	C	D	V	E
Addressing Mode	Assembly	Code	Bytes	Cycles				
(indirect,X)	AND (\$nn,X)	21	2	5	<i>r</i>			
base-page	AND \$nn	25	2	3	<i>r</i>			
immediate	AND #\$nn	29	2	2				
absolute	AND \$nnnn	2D	3	4	<i>r</i>			
(indirect),Y	AND (\$nn),Y	31	2	5	<i>pr</i>			
(indirect),Z	AND (\$nn),Z	32	2	5	<i>pr</i>			
base-page,X	AND \$nn,X	35	2	3	<i>pr</i>			
absolute,Y	AND \$nnnn,Y	39	3	4	<i>pr</i>			
absolute,X	AND \$nnnn,X	3D	3	4	<i>pr</i>			

*p* Add one cycle if indexing crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

## ASL

This instruction shifts either the Accumulator or contents of the provided memory location one bit left. Bit 0 will be set to zero, and the bit 7 will be shifted out into the Carry Flag

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 7 of the value was set, prior to being shifted.

**ASL : Arithmetic Shift Left****4510** $A \leftarrow A \lll 1$  or  $M \leftarrow M \lll 1$ **N Z I C D V E**

+ + . + . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
base-page	ASL \$nn	06	2	4	<i>dmr</i>
accumulator	ASL A	0A	1	1	<i>s</i>
absolute	ASL \$nnnn	0E	3	5	<i>dmr</i>
base-page,X	ASL \$nn,X	16	2	4	<i>dmpr</i>
absolute,X	ASL \$nnnn,X	1E	3	5	<i>dmpr</i>

*d* Subtract one cycle when CPU is at 3.5MHz.*m* Subtract non-bus cycles when at 40MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

# ASR

This instruction shifts either the Accumulator or contents of the provided memory location one bit right. Bit 7 is considered to be a sign bit, and is preserved. The contents of bit 0 will be shifted out into the Carry Flag

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 0 of the value was set, prior to being shifted.

**ASR : Arithmetic Shift Right****4510** $A \leftarrow A >> 1$  or  $M \leftarrow M >> 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
accumulator	ASR A	43	1	2	
base-page	ASR \$nn	44	2	4	<i>dmr</i>
base-page,X	ASR \$nn,X	54	2	4	<i>dmpr</i>

*d* Subtract one cycle when CPU is at 3.5MHz.*m* Subtract non-bus cycles when at 40MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

## ASW

This instruction shifts a 16-bit value in memory left one bit.

For example, if location \$1234 contained \$87 and location \$1235 contained \$A9, ASW \$1234 would result in location \$1234 containing \$0E and location \$1235 containing \$53, and the Carry Flag being set.

### Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 of the upper byte is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 7 of the upper byte was set, prior to being shifted.

**ASW : Arithmetic Shift Word Left****4510** $M \leftarrow M << 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
absolute	ASW \$nnnn	CB	3	7	<i>dmr</i>

*d* Subtract one cycle when CPU is at 3.5MHz.*m* Subtract non-bus cycles when at 40MHz.*r* Add one cycle if clock speed is at 40 MHz.

## BBR0

This instruction branches to the indicated address if bit 0 is clear in the indicated base-page memory location.

<b>BBR0 : Branch on Bit 0 Reset</b>		4510		
PC ← PC + R8		N Z I C D V E		
		. . . . .		
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBR0 \$nn,\$rr	0F	3	4 <i>br</i>

- b Add one cycle if branch is taken.
- Add one more cycle if branch taken crosses a page boundary.
- r Add one cycle if clock speed is at 40 MHz.

## BBR1

This instruction branches to the indicated address if bit 1 is clear in the indicated base-page memory location.

<b>BBR1 : Branch on Bit 1 Reset</b>		4510		
PC ← PC + R8		N Z I C D V E		
		. . . . .		
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBR1 \$nn,\$rr	1F	3	4 <i>br</i>

- b Add one cycle if branch is taken.
- Add one more cycle if branch taken crosses a page boundary.
- r Add one cycle if clock speed is at 40 MHz.

## BBR2

This instruction branches to the indicated address if bit 2 is clear in the indicated base-page memory location.

**BBR2 : Branch on Bit 2 Reset**

4510

PC  $\leftarrow$  PC + R8

N Z I C D V E

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBR2 \$nn,\$rr	2F	3	4 <i>br</i>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.**BBR3**

This instruction branches to the indicated address if bit 3 is clear in the indicated base-page memory location.

**BBR3 : Branch on Bit 3 Reset**

4510

PC  $\leftarrow$  PC + R8

N Z I C D V E

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBR3 \$nn,\$rr	3F	3	4 <i>br</i>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.**BBR4**

This instruction branches to the indicated address if bit 4 is clear in the indicated base-page memory location.

**BBR4 : Branch on Bit 4 Reset**

4510

PC  $\leftarrow$  PC + R8

N Z I C D V E

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBR4 \$nn,\$rr	4F	3	4 <i>br</i>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

## BBR5

This instruction branches to the indicated address if bit 5 is clear in the indicated base-page memory location.

<b>BBR5 : Branch on Bit 5 Reset</b>		4510		
PC ← PC + R8		N	Z	I C D V E
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBR5 \$nn,\$rr	5F	3	4 <i>br</i>

- b Add one cycle if branch is taken.
- Add one more cycle if branch taken crosses a page boundary.
- r Add one cycle if clock speed is at 40 MHz.

## BBR6

This instruction branches to the indicated address if bit 6 is clear in the indicated base-page memory location.

<b>BBR6 : Branch on Bit 6 Reset</b>		4510		
PC ← PC + R8		N	Z	I C D V E
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBR6 \$nn,\$rr	6F	3	4 <i>br</i>

- b Add one cycle if branch is taken.
- Add one more cycle if branch taken crosses a page boundary.
- r Add one cycle if clock speed is at 40 MHz.

## BBR7

This instruction branches to the indicated address if bit 7 is clear in the indicated base-page memory location.

**BBR7 : Branch on Bit 7 Reset**

4510

PC ← PC + R8

N Z I C D V E

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBR7 \$nn,\$rr	7F	3	4 <i>br</i>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

**BBS0**

This instruction branches to the indicated address if bit 0 is set in the indicated base-page memory location.

**BBS0 : Branch on Bit 0 Set**

4510

PC ← PC + R8

N Z I C D V E

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBS0 \$nn,\$rr	8F	3	4 <i>br</i>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

**BBS1**

This instruction branches to the indicated address if bit 1 is set in the indicated base-page memory location.

**BBS1 : Branch on Bit 1 Set**

4510

PC ← PC + R8

N Z I C D V E

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBS1 \$nn,\$rr	9F	3	4 <i>br</i>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

## BBS2

This instruction branches to the indicated address if bit 2 is set in the indicated base-page memory location.

<b>BBS2 : Branch on Bit 2 Set</b>		4510		
PC ← PC + R8		N Z I C D V E		
		. . . . .		
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBS2 \$nn,\$rr	AF	3	4 <i>br</i>

- b Add one cycle if branch is taken.
- Add one more cycle if branch taken crosses a page boundary.
- r Add one cycle if clock speed is at 40 MHz.

## BBS3

This instruction branches to the indicated address if bit 3 is set in the indicated base-page memory location.

<b>BBS3 : Branch on Bit 3 Set</b>		4510		
PC ← PC + R8		N Z I C D V E		
		. . . . .		
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBS3 \$nn,\$rr	BF	3	4 <i>br</i>

- b Add one cycle if branch is taken.
- Add one more cycle if branch taken crosses a page boundary.
- r Add one cycle if clock speed is at 40 MHz.

## BBS4

This instruction branches to the indicated address if bit 4 is set in the indicated base-page memory location.

**BBS4 : Branch on Bit 4 Set**

4510

PC  $\leftarrow$  PC + R8

N Z I C D V E

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBS4 \$nn,\$rr	CF	3	4 <i>br</i>

b Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

r Add one cycle if clock speed is at 40 MHz.

**BBS5**

This instruction branches to the indicated address if bit 5 is set in the indicated base-page memory location.

**BBS5 : Branch on Bit 5 Set**

4510

PC  $\leftarrow$  PC + R8

N Z I C D V E

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBS5 \$nn,\$rr	DF	3	4 <i>br</i>

b Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

r Add one cycle if clock speed is at 40 MHz.

**BBS6**

This instruction branches to the indicated address if bit 6 is set in the indicated base-page memory location.

**BBS6 : Branch on Bit 6 Set**

4510

PC  $\leftarrow$  PC + R8

N Z I C D V E

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
base-page and relative	BBS6 \$nn,\$rr	EF	3	4 <i>br</i>

b Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

r Add one cycle if clock speed is at 40 MHz.

## BBS7

This instruction branches to the indicated address if bit 7 is set in the indicated base-page memory location.

<b>BBS7 : Branch on Bit 7 Set</b>		4510		
PC ← PC + R8		N	Z	I C D V E
Addressing Mode	Assembly	Code	Bytes	Cycles
base-page and relative	BBS7 \$nn,\$rr	FF	3	4 <sup>b</sup>

- b* Add one cycle if branch is taken.  
Add one more cycle if branch taken crosses a page boundary.
- r* Add one cycle if clock speed is at 40 MHz.

## BCC

This instruction branches to the indicated address if the Carry Flag is clear.

<b>BCC : Branch on Carry Flag Clear</b>		4510		
PC ← PC + R8 or PC ← PC + R16		N	Z	I C D V E
Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BCC \$rr	90	2	2 <sup>b</sup>
16-bit relative	BCC \$rrrr	93	3	3 <sup>b</sup>

- b* Add one cycle if branch is taken.  
Add one more cycle if branch taken crosses a page boundary.

## BCS

This instruction branches to the indicated address if the Carry Flag is set.

**BCS : Branch on Carry Flag Set****4510**PC  $\leftarrow$  PC + R8 or PC  $\leftarrow$  PC + R16

N Z I C D V E

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
relative	BCS \$rr	B0	2	2
16-bit relative	BCS \$rrrr	B3	3	3

b Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

**BEQ**

This instruction branches to the indicated address if the Zero Flag is set.

**BEQ : Branch on Zero Flag Set****4510**PC  $\leftarrow$  PC + R8 or PC  $\leftarrow$  PC + R16

N Z I C D V E

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
relative	BEQ \$rr	F0	2	2
16-bit relative	BEQ \$rrrr	F3	3	3

b Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

**BIT**

This instruction is used to test the bits stored in a memory location. Bits 6 and 7 of the memory location's contents are directly copied into the Overflow Flag and Negative Flag. The Zero Flag is set or cleared based on the result of performing the binary AND of the Accumulator Register and the contents of the indicated memory location.

**Side effects**

- The N flag will be set if the bit 7 of the memory location is set, else it will be cleared.
- The V flag will be set if the bit 6 of the memory location is set, else it will be cleared.

- The Z flag will be set if the result of A AND M is zero, else it will be cleared.

<b>BIT : Perform Bit Test</b>		<b>4510</b>			
N $\leftarrow$ M(7), V $\leftarrow$ M(6), Z $\leftarrow$ A AND M					
		N Z I C D V E			
		+ + . . . + .			
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
base-page	BIT \$nn	24	2	3	<i>r</i>
absolute	BIT \$nnnn	2C	3	4	<i>r</i>
base-page,X	BIT \$nn,X	34	2	3	<i>pr</i>
absolute,X	BIT \$nnnn,X	3C	3	4	<i>pr</i>
immediate	BIT #\$nn	89	2	2	

*p* Add one cycle if indexing crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

## BMI

This instruction branches to the indicated address if the Negative Flag is set.

<b>BMI : Branch on Negative Flag Set</b>		<b>4510</b>			
PC $\leftarrow$ PC + R8 or PC $\leftarrow$ PC + R16					
		N Z I C D V E			
		. . . . .			
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
relative	BMI \$rr	30	2	2	<i>b</i>
16-bit relative	BMI \$rrrr	33	3	3	<i>b</i>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

## BNE

This instruction branches to the indicated address if the Zero Flag is clear.

**BNE : Branch on Zero Flag Clear****4510**PC  $\leftarrow$  PC + R8 or PC  $\leftarrow$  PC + R16

N Z I C D V E

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
relative	BNE \$rr	D0	2	2
16-bit relative	BNE \$rrrr	D3	3	3

b Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

**BPL**

This instruction branches to the indicated address if the Negative Flag is clear.

**BPL : Branch on Negative Flag Clear****4510**PC  $\leftarrow$  PC + R8 or PC  $\leftarrow$  PC + R16

N Z I C D V E

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
relative	BPL \$rr	10	2	2
16-bit relative	BPL \$rrrr	13	3	3

b Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

**BRA**

This instruction branches to the indicated address.

**BRA : Branch Unconditionally****4510**PC  $\leftarrow$  PC + R8 or PC  $\leftarrow$  PC + R16

N Z I C D V E

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
relative	BRA \$rr	80	2	2
16-bit relative	BRA \$rrrr	83	3	3

b Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

# BRK

The break command causes the microprocessor to go through an interrupt sequence under program control. The address of the BRK instruction + 2 is pushed to the stack along with the status register with the Break flag set. This allows the interrupt service routine to distinguish between IRQ events and BRK events. For example:

```
PLA          ; load status
PHA          ; restore stack
AND #$10    ; mask break flag
BNE DO_BREAK ; -> it was a BRK
...          ; else continue with IRQ server
```

Cite from: MCS6500 Microcomputer Family Programming Manual, January 1976, Second Edition, MOS Technology Inc., Page 144:

"The BRK is a single byte instruction and its addressing mode is Implied."

There are debates, that BRK could be seen as a two byte instruction with the addressing mode immediate, where the operand byte is discarded. The byte following the BRK could then be used as a call argument for the break handler. Commodore however used the BRK, as stated in the manual, as a single byte instruction, which breaks into the ML monitor, if present. These builtin monitors decremented the stacked PC, so that it could be used to return or jump directly to the code byte after the BRK.

BRK : Break to Interrupt	4510
PC ← (\$FFFE)	
	N Z I C D V E
	. . . . .
Addressing Mode	Assembly
implied	BRK
Code	Bytes
00	1
Cycles	7

# BSR

This instruction branches to the indicated address, saving the address of the caller on the stack, so that the routine can be returned from using an RTS instruction.

This instruction is helpful for using relocatable code, as it provides a relative-addressed alternative to JSR.

**BSR : Branch Sub-Routine****4510**PC  $\leftarrow$  PC + R8 or PC  $\leftarrow$  PC + R16

N Z I C D V E

**Addressing Mode****Assembly Code****Bytes****Cycles**

16-bit relative

BSR \$rrrr

63

3

3

<sup>b</sup>*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

## BVC

This instruction branches to the indicated address if the Overflow (V) Flag is clear.

**BVC : Branch on Overflow Flag Clear****4510**PC  $\leftarrow$  PC + R8 or PC  $\leftarrow$  PC + R16

N Z I C D V E

**Addressing Mode****Assembly Code****Bytes****Cycles**

relative

BVC \$rr

50

2

2

<sup>b</sup>

16-bit relative

BVC \$rrrr

53

3

3

<sup>b</sup>*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

## BVS

This instruction branches to the indicated address if the Overflow (V) Flag is set.

**BVS : Branch on Overflow Flag Set****4510**PC  $\leftarrow$  PC + R8 or PC  $\leftarrow$  PC + R16

N Z I C D V E

**Addressing Mode****Assembly Code****Bytes****Cycles**

relative

BVS \$rr

70

2

2

<sup>b</sup>

16-bit relative

BVS \$rrrr

73

3

3

<sup>b</sup>*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

# CLC

This instruction clears the Carry Flag.

## Side effects

- The C flag is cleared.

CLC : Clear Carry Flag		4510				
C ← 0		N	Z	I	C	D V E
		.	.	.	.	.
Addressing Mode	Assembly	Code	Bytes	Bytes	Cycles	
implied	CLC	18	1	1	<sup>s</sup>	

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

# CLD

This instruction clears the Decimal Flag. Arithmetic operations will use normal binary arithmetic, instead of Binary-Coded Decimal (BCD).

## Side effects

- The D flag is cleared.

CLD : Clear Decimal Flag		4510				
D ← 0		N	Z	I	C	D V E
		.	.	.	.	.
Addressing Mode	Assembly	Code	Bytes	Bytes	Cycles	
implied	CLD	D8	1	1		

# CLE

This instruction clears the Extended Stack Disable Flag. This causes the stack to be able to exceed 256 bytes in length, by allowing the processor to modify the value of the high byte of the stack address (SPH).

## Side effects

- The E flag is cleared.

<b>CLE : Clear Extended Stack Disable Flag</b>		<b>4510</b>
E ← 0		
		N Z I C D V E
		· · · · ·
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
implied	CLE	02
		1
		1
		<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## CLI

This instruction clears the Interrupt Disable Flag. Interrupts will now be able to occur.

## Side effects

- The I flag is cleared.

<b>CLI : Clear Interrupt Disable Flag</b>		<b>4510</b>
I ← 0		
		N Z I C D V E
		· · · · ·
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
implied	CLI	58
		1
		1
		<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## CLV

This instruction clears the Overflow Flag.

## Side effects

- The V flag is cleared.

<b>CLV : Clear Overflow Flag</b>	<b>4510</b>				
V ← 0					
N Z I C D V E					
	.				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	CLV		B8	1	

## CMP

This instruction performs A – M, and sets the processor flags accordingly, but does not modify the contents of the Accumulator Register.

### Side effects

- The N flag will be set if the result of A – M is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of A – M is zero or positive, i.e., if A is not less than M, else it will be cleared.
- The Z flag will be set if the result of A – M is zero, else it will be cleared.

<b>CMP : Compare Accumulator</b>	<b>4510</b>
N Z I C D V E	
	.
<b>Addressing Mode</b>	<b>Assembly</b>
(indirect,X)	CMP (\$nn,X)
base-page	CMP \$nn
immediate	CMP #\$nn
absolute	CMP \$nnnn
(indirect),Y	CMP (\$nn),Y
(indirect),Z	CMP (\$nn),Z
base-page,X	CMP \$nn,X
absolute,Y	CMP \$nnnn,Y
absolute,X	CMP \$nnnn,X

*p* Add one cycle if indexing crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

# CPX

This instruction performs  $X - M$ , and sets the processor flags accordingly, but does not modify the contents of the Accumulator Register.

## Side effects

- The N flag will be set if the result of  $X - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of  $X - M$  is zero or positive, i.e., if  $X$  is not less than  $M$ , else it will be cleared.
- The Z flag will be set if the result of  $X - M$  is zero, else it will be cleared.

CPX : Compare X Register		4510						
		N	Z	I	C	D	V	E
		+	+	.	+	.	.	.
Addressing Mode		Assembly	Code	Bytes	Cycles			
immediate		CPX #\$nn	E0	2	2			
base-page		CPX \$nn	E4	2	3			
absolute		CPX \$nnnn	EC	3	4			

r Add one cycle if clock speed is at 40 MHz.

# CPY

This instruction performs  $Y - M$ , and sets the processor flags accordingly, but does not modify the contents of the Accumulator Register.

## Side effects

- The N flag will be set if the result of  $Y - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of  $Y - M$  is zero or positive, i.e., if  $Y$  is not less than  $M$ , else it will be cleared.
- The Z flag will be set if the result of  $Y - M$  is zero, else it will be cleared.

**CPY : Compare Y Register****4510**

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
immediate	CPY #\$nn	C0	2	2	
base-page	CPY \$nn	C4	2	3	<i>r</i>
absolute	CPY \$nnnn	CC	3	4	<i>r</i>

*r* Add one cycle if clock speed is at 40 MHz.

**CPZ**

This instruction performs  $Z - M$ , and sets the processor flags accordingly, but does not modify the contents of the Accumulator Register.

**Side effects**

- The N flag will be set if the result of  $Z - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of  $Z - M$  is zero or positive, i.e., if  $Z$  is not less than  $M$ , else it will be cleared.
- The Z flag will be set if the result of  $Z - M$  is zero, else it will be cleared.

**CPZ : Compare Z Register****4510**

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
immediate	CPZ #\$nn	C2	2	2	
base-page	CPZ \$nn	D4	2	3	<i>r</i>
absolute	CPZ \$nnnn	DC	3	4	<i>r</i>

*r* Add one cycle if clock speed is at 40 MHz.

**DEC**

This instruction decrements the Accumulator Register or indicated memory location.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

DEC : Decrement Memory or Accumulator		4510						
A $\leftarrow$ A - 1 or M $\leftarrow$ M - 1		N	Z	I	C	D	V	E
		+	+	.	.	.	.	.
Addressing Mode	Assembly	Code	Bytes	Bytes	Cycles			
accumulator	DEC A	3A	1	1	s			
base-page	DEC \$nn	C6	2	5	dmr			
absolute	DEC \$nnnn	CE	3	6	dmr			
base-page,X	DEC \$nn,X	D6	2	5	dmpr			
absolute,X	DEC \$nnnn,X	DE	3	6	dmpr			

*d* Subtract one cycle when CPU is at 3.5MHz.

*m* Subtract non-bus cycles when at 40MHz.

*p* Add one cycle if indexing crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## DEW

This instruction decrements the indicated memory word in the Base Page. The low numbered address contains the least significant bits. For example, if memory location \$12 contains \$78 and memory location \$13 contains \$56, the instruction DEW \$12 would cause memory location to be set to \$77.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**DEW : Decrement Memory Word****4510** $M16 \leftarrow M16 - 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
base-page	DEW \$nn	C3	2	7	<i>dmr</i>

*d* Subtract one cycle when CPU is at 3.5MHz.*m* Subtract non-bus cycles when at 40MHz.*r* Add one cycle if clock speed is at 40 MHz.

## DEX

This instruction decrements the X Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**DEX : Decrement X Register****4510** $X \leftarrow X - 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	DEX	CA	1	1	<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## DEY

This instruction decrements the Y Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>DEY : Decrement Y Register</b>	<b>4510</b>
$Y \leftarrow Y - 1$	
	<b>N Z I C D V E</b> + + . . . .
<b>Addressing Mode</b>	<b>Assembly Code</b>

Addressing Mode	Assembly Code	Bytes	Cycles	
implied	DEY	88	1	1 <i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## DEZ

This instruction decrements the Z Register.

### Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>DEZ : Decrement Z Register</b>	<b>4510</b>
$Z \leftarrow Z - 1$	
	<b>N Z I C D V E</b> + + . . . .
<b>Addressing Mode</b>	<b>Assembly Code</b>

Addressing Mode	Assembly Code	Bytes	Cycles	
implied	DEZ	3B	1	1 <i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## EOM

In contrast with the 6502, the NOP instruction on the 45GS02 performs two additional roles when in 4502 mode.

First, indicate the end of a memory mapping sequence caused by a MAP instruction, allowing interrupts to occur again.

Second, it instructs the processor that if the following instruction uses Base-Page Indirect Z Indexed addressing, that the processor should use a 32-bit pointer instead of a 16-bit 6502 style pointer. Such 32-bit addresses are unaffected by C64, C65 or MEGA65 memory banking. This allows fast and easy access to the entire address space of the MEGA65 without having to perform or be aware of any banking, or using the DMA controller. This ad-

dressing mode causes a two cycle penalty, caused by the time required to read the extra two bytes of the pointer.

### Side effects

- Removes the prohibition on all interrupts caused by the MAP instruction, allowing Non-Maskable Interrupts to again occur, and IRQ interrupts, if the Interrupt Disable Flag is not set.

<b>EOM : End of Mapping Sequence / No-Operation</b>		<b>4510</b>				
		N Z I C D V E				
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>			
implied	EOM	EA	1	1	<i>s</i>	

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## EOR

This instruction performs a binary XOR operation of the argument with the accumulator, and stores the result in the accumulator. Only bits that were already set in the accumulator, or that are set in the argument will be set in the accumulator on completion, but not both.

### Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

A  $\leftarrow$  A XOR M

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	EOR (\$nn,X)	41	2	5	<i>pr</i>
base-page	EOR \$nn	45	2	3	<i>r</i>
immediate	EOR #\$nn	49	2	2	
absolute	EOR \$nnnn	4D	3	4	<i>r</i>
(indirect),Y	EOR (\$nn),Y	51	2	5	<i>pr</i>
(indirect),Z	EOR (\$nn),Z	52	2	5	<i>pr</i>
base-page,X	EOR \$nn,X	55	2	3	<i>pr</i>
absolute,Y	EOR \$nnnn,Y	59	3	4	<i>pr</i>
absolute,X	EOR \$nnnn,X	5D	3	4	<i>pr</i>

*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

## INC

This instruction increments the Accumulator Register or indicated memory location.

### Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**INC : Increment Memory or Accumulator****4510** $A \leftarrow A + 1$  or  $M \leftarrow M + 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
accumulator	INC A	1A	1	1	<i>s</i>
base-page	INC \$nn	E6	2	5	<i>dmr</i>
absolute	INC \$nnnn	EE	3	6	<i>dmr</i>
base-page,X	INC \$nn,X	F6	2	5	<i>dmpr</i>
absolute,X	INC \$nnnn,X	FE	3	6	<i>dpr</i>

*d* Subtract one cycle when CPU is at 3.5MHz.*m* Subtract non-bus cycles when at 40MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.**INW**

This instruction increments the indicated memory word in the Base Page. The low numbered address contains the least significant bits. For example, if memory location \$12 contains \$78 and memory location \$13 contains \$56, the instruction DEW \$12 would cause memory location to be set to \$79.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**INW : Increment Memory Word****4510** $M16 \leftarrow M16 + 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
base-page	INW \$nn	E3	2	7	<i>dmr</i>

*d* Subtract one cycle when CPU is at 3.5MHz.*m* Subtract non-bus cycles when at 40MHz.*r* Add one cycle if clock speed is at 40 MHz.

# INX

This instruction increments the X Register, i.e., adds 1 to it.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

INX : Increment X Register		4510
$X \leftarrow X + 1$		
		N Z I C D V E
		+ + . . . .
Addressing Mode	Assembly Code	Bytes Cycles
implied	INX	E8 1 1 <sup>s</sup>

<sup>s</sup> Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

# INY

This instruction increments the Y Register, i.e., adds 1 to it.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

INY : Increment Y Register		4510
$Y \leftarrow Y + 1$		
		N Z I C D V E
		+ + . . . .
Addressing Mode	Assembly Code	Bytes Cycles
implied	INY	C8 1 1 <sup>s</sup>

<sup>s</sup> Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

# INZ

This instruction increments the Z Register, i.e., adds 1 to it.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>INZ : Increment Z Register</b>	<b>4510</b>			
$Z \leftarrow Y + 1$				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	INZ	1B	1	1 <sup>s</sup>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## JMP

This instruction sets the Programme Counter (PC) Register to the address indicated by the instruction, causing execution to continue from that address.

<b>JMP : Jump to Address</b>	<b>4510</b>			
$PC \leftarrow M2:M1$				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
absolute	JMP \$nnnn	4C	3	3
indirect	JMP (\$nnnn)	6C	3	5 <sup>m</sup>
indirect,X	JMP (\$nnnn,X)	7C	3	6 <sup>mp</sup>

*m* Subtract non-bus cycles when at 40MHz.

*p* Add one cycle if indexing crosses a page boundary.

## JSR

This instruction saves the address of the instruction following the JSR instruction onto the stack, and then sets the Programme Counter (PC) Register to the address indicated by the instruction, causing execution to continue from that address. Because the return address has been saved on the stack, the RTS instruction can be used to return from the called sub-routine and resume execution following the JSR instruction.

NOTE: This instruction actually pushes the address of the last byte of the JSR instruction onto the stack. The RTS instruction naturally is aware of this, and

increments the address on popping it from the stack, before setting the Programme Counter (PC) register.

<b>JSR : Jump to Sub-Routine</b>		<b>4510</b>				
PC $\leftarrow$ M2:M1, Stack $\leftarrow$ PCH:PCL		N Z I C D V E				
		.	.	.	.	.
<b>Addressing Mode</b>		<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
absolute		JSR \$nnnn	20	3	5	
indirect		JSR (\$nnnn)	22	3	7	
indirect,X		JSR (\$nnnn,X)	23	3	7	

## LDA

This instruction loads the Accumulator Register with the indicated value, or with the contents of the indicated location.

### Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>LDA : Load Accumulator</b>		<b>4510</b>				
A $\leftarrow$ M		N Z I C D V E				
		.	.	.	.	.
<b>Addressing Mode</b>		<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)		LDA (\$nn,X)	A1	2	5	<i>pr</i>
base-page		LDA \$nn	A5	2	3	<i>r</i>
immediate		LDA #\$nn	A9	2	2	
absolute		LDA \$nnnn	AD	3	4	<i>r</i>
(indirect),Y		LDA (\$nn),Y	B1	2	5	<i>pr</i>
(indirect),Z		LDA (\$nn),Z	B2	2	5	<i>pr</i>
base-page,X		LDA \$nn,X	B5	2	3	<i>pr</i>
absolute,Y		LDA \$nnnn,Y	B9	3	4	<i>pr</i>
absolute,X		LDA \$nnnn,X	BD	3	4	<i>pr</i>
(indirect,SP),Y		LDA (\$nn,SP),Y	E2	2	6	<i>mpr</i>

*m* Subtract non-bus cycles when at 40MHz.

*p* Add one cycle if indexing crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

# LDX

This instruction loads the X Register with the indicated value, or with the contents of the indicated location.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

LDX : Load X Register		4510						
X ← M		N	Z	I	C	D	V	E
		+	+	.	.	.	.	.
Addressing Mode	Assembly	Code	Bytes					Cycles
immediate	LDX #\$nn	A2	2	2				
base-page	LDX \$nn	A6	2	3	r			
absolute	LDX \$nnnn	AE	3	4	r			
base-page,Y	LDX \$nn,Y	B6	2	5	pr			
absolute,Y	LDX \$nnnn,Y	BE	3	4	pr			

p Add one cycle if indexing crosses a page boundary.

r Add one cycle if clock speed is at 40 MHz.

# LDY

This instruction loads the Y Register with the indicated value, or with the contents of the indicated location.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**LDY : Load Y Register**

4510

 $Y \leftarrow M$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
immediate	LDY #\$nn	A0	2	2	
base-page	LDY \$nn	A4	2	3	<i>r</i>
absolute	LDY \$nnnn	AC	3	4	<i>r</i>
base-page,X	LDY \$nn,X	B4	2	3	<i>pr</i>
absolute,X	LDY \$nnnn,X	BC	3	4	<i>pr</i>

*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.**LDZ**

This instruction loads the Z Register with the indicated value, or with the contents of the indicated location.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**LDZ : Load Z Register**

4510

 $Z \leftarrow M$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
immediate	LDZ #\$nn	A3	2	2	
absolute	LDZ \$nnnn	AB	3	4	<i>r</i>
absolute,X	LDZ \$nnnn,X	BB	3	4	<i>pr</i>

*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.**LSR**

This instruction shifts either the Accumulator or contents of the provided memory location one bit right. Bit 7 will be set to zero, and the bit 0 will be shifted out into the Carry Flag

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 0 of the value was set, prior to being shifted.

LSR : Logical Shift Right						4510		
$A \leftarrow A >> 1, C \leftarrow A(0) \text{ or } M \leftarrow M >> 1$								
		N	Z	I	C	D	V	E
		+	+	.	+	.	.	.
Addressing Mode	Assembly	Code	Bytes		Cycles			
base-page	LSR \$nn	46	2	5		<i>dmr</i>		
accumulator	LSR A	4A	1	1		<i>s</i>		
absolute	LSR \$nnnn	4E	3	5		<i>dmr</i>		
base-page,X	LSR \$nn,X	56	2	5		<i>dmpr</i>		
absolute,X	LSR \$nnnn,X	5E	3	6		<i>dmpr</i>		

*d* Subtract one cycle when CPU is at 3.5MHz.

*m* Subtract non-bus cycles when at 40MHz.

*p* Add one cycle if indexing crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## MAP

This instruction sets the C65 or MEGA65 style memory map, depending on the values in the Accumulator, X, Y and Z registers.

Care should be taken to ensure that after the execution of an MAP instruction that appropriate memory is mapped at the location of the following instruction. Failure to do so will result in unpredictable results.

Further information on this instruction is available in Appendix G.

## Side effects

- The memory map is immediately changed to that requested.
- All interrupts, including Non-Maskable Interrupts (NMIs) are blocked from occurring until an EOM (NOP) instruction is encountered.

**MAP : Set Memory Map****4510**

N Z I C D V E

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	MAP	5C	1	1 <sup>s</sup>

<sup>s</sup> Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## NEG

This instruction replaces the contents of the Accumulator Register with the two's-complement of the contents of the Accumulator Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**NEG : Negate Accumulator****4510**A  $\leftarrow$  -AN Z I C D V E  
+ + . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
accumulator	NEG A	42	1	2 <sup>m</sup>

<sup>m</sup> Subtract non-bus cycles when at 40MHz.

## ORA

This instruction performs a binary OR operation of the argument with the accumulator, and stores the result in the accumulator. Only bits that were already set in the accumulator, or that are set in the argument will be set in the accumulator on completion, or both.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**ORA : Decrement Memory or Accumulator****4510** $A \leftarrow A + 1$  or  $M \leftarrow M + 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	ORA (\$nn,X)	01	2	5	<i>pr</i>
base-page	ORA \$nn	05	2	3	<i>r</i>
immediate	ORA #\$nn	09	2	2	
absolute	ORA \$nnnn	0D	3	4	<i>r</i>
(indirect),Y	ORA (\$nn),Y	11	2	5	<i>pr</i>
(indirect),Z	ORA (\$nn),Z	12	2	5	<i>pr</i>
base-page,X	ORA \$nn,X	15	2	3	<i>pr</i>
absolute,Y	ORA \$nnnn,Y	19	3	4	<i>pr</i>
absolute,X	ORA \$nnnn,X	1D	3	4	<i>pr</i>

*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

## PHA

This instruction pushes the contents of the Accumulator Register onto the stack, and decrements the value of the Stack Pointer by 1.

**PHA : Push Accumulator Register onto the Stack****4510** $STACK \leftarrow A$ ,  $SP \leftarrow SP - 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
.	.	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	PHA	48	1	3	<i>m</i>

*m* Subtract non-bus cycles when at 40MHz.

## PHP

This instruction pushes the contents of the Processor Flags onto the stack, and decrements the value of the Stack Pointer by 1.

**PHP : Push Processor Flags onto the Stack****4510**STACK  $\leftarrow$  P, SP  $\leftarrow$  SP - 1**N Z I C D V E****Addressing Mode****Assembly Code****Bytes****Cycles**

implied

PHP

08

1

3

*m**m* Subtract non-bus cycles when at 40MHz.

## PHW

This instruction pushes either a 16 bit literal value or the memory word indicated onto the stack, and decrements the value of the Stack Pointer by 2.

**PHW : Push Word onto the Stack****4510**STACK  $\leftarrow$  M1:M2, SP  $\leftarrow$  SP - 2**N Z I C D V E**  
+ + . . . .**Addressing Mode****Assembly****Code****Bytes****Cycles**

immediate

PHW #\$nnnn

F4

2

5

*m*

absolute

PHW \$nnnn

FC

3

7

*m**m* Subtract non-bus cycles when at 40MHz.

## PHX

This instruction pushes the contents of the X Register onto the stack, and decrements the value of the Stack Pointer by 1.

**PHX : Push X Register onto the Stack****4510**STACK  $\leftarrow$  X, SP  $\leftarrow$  SP - 1**N Z I C D V E**  
. . . . . .**Addressing Mode****Assembly****Code****Bytes****Cycles**

implied

PHX

DA

1

3

*m**m* Subtract non-bus cycles when at 40MHz.

# PHY

This instruction pushes the contents of the Y Register onto the stack, and decrements the value of the Stack Pointer by 1.

<b>PHY : Push Y Register onto the Stack</b>		<b>4510</b>	
STACK $\leftarrow$ Y, SP $\leftarrow$ SP - 1			
N Z I C D V E			
. . . . .			
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	PHY	5A	1      3 <i>m</i>

*m* Subtract non-bus cycles when at 40MHz.

# PHZ

This instruction pushes the contents of the Z Register onto the stack, and decrements the value of the Stack Pointer by 1.

<b>PHZ : Push Z Register onto the Stack</b>		<b>4510</b>	
STACK $\leftarrow$ z, SP $\leftarrow$ SP - 1			
N Z I C D V E			
. . . . .			
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	PHZ	DB	1      3 <i>m</i>

*m* Subtract non-bus cycles when at 40MHz.

# PLA

This instruction replaces the contents of the Accumulator Register with the top value from the stack, and increments the value of the Stack Pointer by 1.

<b>PLA : Pull Accumulator Register from the Stack</b>		<b>4510</b>	
A $\leftarrow$ STACK, SP $\leftarrow$ SP + 1			
N Z I C D V E			
+ + . . . .			
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	PLA	68	1      3 <i>m</i>

*m* Subtract non-bus cycles when at 40MHz.

## PLP

This instruction replaces the contents of the Processor Flags with the top value from the stack, and increments the value of the Stack Pointer by 1.

NOTE: This instruction does NOT replace the Extended Stack Disable Flag (E Flag), or the Software Interrupt Flag (B Flag)

<b>PLP : Pull Accumulator Register from the Stack</b>		<b>4510</b>	
A $\leftarrow$ STACK, SP $\leftarrow$ SP + 1			
		N Z I C D V E	
		+ + + + + + .	
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	PLP	28	1 3 <i>m</i>

*m* Subtract non-bus cycles when at 40MHz.

## PLX

This instruction replaces the contents of the X Register with the top value from the stack, and increments the value of the Stack Pointer by 1.

<b>PLX : Pull X Register from the Stack</b>		<b>4510</b>	
X $\leftarrow$ STACK, SP $\leftarrow$ SP + 1			
		N Z I C D V E	
		+ + . . . .	
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	PLX	FA	1 3 <i>m</i>

*m* Subtract non-bus cycles when at 40MHz.

## PLY

This instruction replaces the contents of the Y Register with the top value from the stack, and increments the value of the Stack Pointer by 1.

**PLY : Pull Y Register from the Stack****4510** $Y \leftarrow \text{STACK}, SP \leftarrow SP + 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

**Addressing Mode****Assembly Code****Bytes****Cycles**

implied

PLY

7A

1

3

*m**m* Subtract non-bus cycles when at 40MHz.

## PLZ

This instruction replaces the contents of the Z Register with the top value from the stack, and increments the value of the Stack Pointer by 1.

**PLZ : Pull Z Register from the Stack****4510** $Z \leftarrow \text{STACK}, SP \leftarrow SP + 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

**Addressing Mode****Assembly Code****Bytes****Cycles**

implied

PLZ

FB

1

3

*m**m* Subtract non-bus cycles when at 40MHz.

## RMBO

This instruction clears bit zero of the indicated address. No flags are modified, regardless of the result.

**RMBO : Reset Bit 0 in Base Page****4510** $M(0) \leftarrow 0$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

**Addressing Mode****Assembly Code****Bytes****Cycles**

base-page

RMBO \$nn

07

2

4

*r**r* Add one cycle if clock speed is at 40 MHz.

## RMB1

This instruction clears bit 1 of the indicated address. No flags are modified, regardless of the result.

**RMB1 : Reset Bit 1 in Base Page** 4510 $M(1) \leftarrow 0$ 

N Z I C D V E

. . . . .

**Addressing Mode**    **Assembly Code**    **Bytes**    **Cycles**base-page              RMB1 \$nn    17    2    4    *r**r* Add one cycle if clock speed is at 40 MHz.

## RMB2

This instruction clears bit 2 of the indicated address. No flags are modified, regardless of the result.

**RMB2 : Reset Bit 2 in Base Page** 4510 $M(2) \leftarrow 0$ 

N Z I C D V E

. . . . .

**Addressing Mode**    **Assembly Code**    **Bytes**    **Cycles**base-page              RMB2 \$nn    27    2    4    *r**r* Add one cycle if clock speed is at 40 MHz.

## RMB3

This instruction clears bit 3 of the indicated address. No flags are modified, regardless of the result.

**RMB3 : Reset Bit 3 in Base Page** 4510 $M(3) \leftarrow 0$ 

N Z I C D V E

. . . . .

**Addressing Mode**    **Assembly Code**    **Bytes**    **Cycles**base-page              RMB3 \$nn    37    2    4    *r**r* Add one cycle if clock speed is at 40 MHz.

## RMB4

This instruction clears bit 4 of the indicated address. No flags are modified, regardless of the result.

**RMB4 : Reset Bit 4 in Base Page****4510** $M(4) \leftarrow 0$ **N Z I C D V E**

. . . . .

**Addressing Mode****Assembly****Code****Bytes****Cycles**

base-page

RMB4

\$nn

47

2

4

*r*

r Add one cycle if clock speed is at 40 MHz.

## RMB5

This instruction clears bit 5 of the indicated address. No flags are modified, regardless of the result.

**RMB5 : Reset Bit 5 in Base Page****4510** $M(5) \leftarrow 0$ **N Z I C D V E**

. . . . .

**Addressing Mode****Assembly****Code****Bytes****Cycles**

base-page

RMB5

\$nn

57

2

4

*r*

r Add one cycle if clock speed is at 40 MHz.

## RMB6

This instruction clears bit 6 of the indicated address. No flags are modified, regardless of the result.

**RMB6 : Reset Bit 6 in Base Page****4510** $M(6) \leftarrow 0$ **N Z I C D V E**

. . . . .

**Addressing Mode****Assembly****Code****Bytes****Cycles**

base-page

RMB6

\$nn

67

2

4

*r*

r Add one cycle if clock speed is at 40 MHz.

## RMB7

This instruction clears bit 7 of the indicated address. No flags are modified, regardless of the result.

**RMB7 : Reset Bit 7 in Base Page****4510** $M(7) \leftarrow 0$ **N Z I C D V E**

. . . . .

**Addressing Mode****Assembly Code****Bytes****Cycles**

base-page

RMB7

\$nnn

77

2

4

r

r Add one cycle if clock speed is at 40 MHz.

# ROL

This instruction shifts either the Accumulator or contents of the provided memory location one bit left. Bit 0 will be set to the current value of the Carry Flag, and the bit 7 will be shifted out into the Carry Flag

**Side effects**

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 7 of the value was set, prior to being shifted.

**ROL : Rotate Left Memory or Accumulator****4510** $M \leftarrow M << 1, C \leftarrow M(7), M(0) \leftarrow C$ **N Z I C D V E**

+ + . + . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
base-page	ROL \$nnn	26	2	5	dmr
accumulator	ROL A	2A	1	1	s
absolute	ROL \$nnnn	2E	3	5	dmr
base-page,X	ROL \$nn,X	36	2	5	dmpr
absolute,X	ROL \$nnnn,X	3E	3	5	dmpr

d Subtract one cycle when CPU is at 3.5MHz.

m Subtract non-bus cycles when at 40MHz.

p Add one cycle if indexing crosses a page boundary.

r Add one cycle if clock speed is at 40 MHz.

s Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

# ROR

This instruction shifts either the Accumulator or contents of the provided memory location one bit right. Bit 7 will be set to the current value of the Carry Flag, and the bit 0 will be shifted out into the Carry Flag

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 7 of the value was set, prior to being shifted.

ROR : Rotate Right Memory or Accumulator		4510								
		M $\leftarrow$ M>>1, C $\leftarrow$ M(0), M(7) $\leftarrow$ C		N	Z	I	C	D	V	E
		+	+	.	+	.	.	.	.	
Addressing Mode	Assembly	Code	Bytes	Cycles						
base-page	ROR \$nn	66	2	5	<i>dmr</i>					
accumulator	ROR A	6A	1	1	<i>s</i>					
absolute	ROR \$nnnn	6E	3	5	<i>dmr</i>					
base-page,X	ROR \$nn,X	76	2	5	<i>dmpr</i>					
absolute,X	ROR \$nnnn,X	7E	3	5	<i>dmpr</i>					

*d* Subtract one cycle when CPU is at 3.5MHz.

*m* Subtract non-bus cycles when at 40MHz.

*p* Add one cycle if indexing crosses a page boundary.

*r* Add one cycle if clock speed is at 40 MHz.

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

# ROW

This instruction rotates the contents of the indicated memory word one bit left. Bit 0 of the low byte will be set to the current value of the Carry Flag, and the bit 7 of the high byte will be shifted out into the Carry Flag

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

- The C flag will be set if bit 7 of the upper byte was set, prior to being shifted.

<b>ROW : Rotate Word Left</b>	<b>4510</b>			
$M2:M1 \leftarrow M2:M1 << 1, C \leftarrow M2(7), M1(0) \leftarrow C$				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
absolute	ROW \$nnnn	EB	3	5 <i>dmr</i>

*d* Subtract one cycle when CPU is at 3.5MHz.

*m* Subtract non-bus cycles when at 40MHz.

*r* Add one cycle if clock speed is at 40 MHz.

## RTI

This instruction pops the processor flags from the stack, and then pops the Programme Counter (PC) register from the stack, allowing an interrupted programme to resume.

- The 6502 Processor Flags are restored from the stack.
- Neither the B (Software Interrupt) nor E (Extended Stack) flags are set by this instruction.

<b>RTI : Return From Interrupt</b>	<b>4510</b>			
$P \leftarrow \text{STACK}, PC \leftarrow \text{STACK}, SP \leftarrow SP + 3$				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	RTI	40	1	5

## RTS

This instruction adds optional argument to the Stack Pointer (SP) Register, and then pops the Programme Counter (PC) register from the stack, allowing a routine to return to its caller.

**RTS : Return From Subroutine****4510**PC  $\leftarrow$  STACK + N, SP  $\leftarrow$  SP + 2 + N**N Z I C D V E**

. . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	RTS	60	1	4
immediate	RTS #\$nn	62	2	7 <sup>d</sup>

*d* Subtract one cycle when CPU is at 3.5MHz.

## SBC

This instruction performs  $A - M - 1 + C$ , and sets the processor flags accordingly. The result is stored in the Accumulator Register.

NOTE: This instruction is affected by the status of the Decimal Flag.

### Side effects

- The N flag will be set if the result of  $A - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of  $A - M$  is zero or positive, i.e., if A is not less than M, else it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, else it will be cleared. If the flag is set, this indicates that a signed overflow has occurred.
- The Z flag will be set if the result of  $A - M$  is zero, else it will be cleared.

**SBC : Subtract With Carry****4510** $A \leftarrow -M - 1 + C$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	+	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	SBC (\$nn,X)	E1	2	3	<i>mp</i>
base-page	SBC \$nn	E5	2	3	<i>r</i>
immediate	SBC #\$nn	E9	2	2	
absolute	SBC \$nnnn	ED	3	4	<i>r</i>
(indirect),Y	SBC (\$nn),Y	F1	2	3	<i>pr</i>
(indirect),Z	SBC (\$nn),Z	F2	2	3	<i>pr</i>
base-page,X	SBC \$nn,X	F5	2	3	<i>pr</i>
absolute,Y	SBC \$nnnn,Y	F9	3	4	<i>pr</i>
absolute,X	SBC \$nnnn,X	FD	3	4	<i>pr</i>

*m* Subtract non-bus cycles when at 40MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

## SEC

This instruction sets the Carry Flag.

**Side effects**

- The C flag is set.

**SEC : Set Carry Flag****4510** $C \leftarrow 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
.	.	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	SEC	38	1	1	<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## SED

This instruction sets the Decimal Flag. Binary arithmetic will now use Binary-Coded Decimal (BCD) mode.

NOTE: The C64's interrupt handler does not clear the Decimal Flag, which makes it dangerous to set the Decimal Flag without first setting the Interrupt Disable Flag.

### Side effects

- The D flag is set.

SED : Set Decimal Flag		4510				
D ← 1		N Z I C D V E				
		. . . . .				
Addressing Mode		Assembly	Code	Bytes	Cycles	
implied		SED	F8	1	1	<sup>s</sup>

<sup>s</sup> Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## SEE

This instruction sets the Extended Stack Disable Flag. This causes the stack to operate as on the 6502, i.e., limited to a single page of memory. The page of memory in which the stack is located can still be modified by setting the Stack Pointer High (SPH) Register.

### Side effects

- The E flag is set.

SEE : Set Extended Stack Disable Flag		4510				
E ← 1		N Z I C D V E				
		. . . . .				
Addressing Mode		Assembly	Code	Bytes	Cycles	
implied		SEE	03	1	1	<sup>s</sup>

<sup>s</sup> Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## SEI

This instruction sets the Interrupt Disable Flag. Normal (IRQ) interrupts will no longer be able to occur. Non-Maskable Interrupts (NMI) will continue to occur, as their name suggests.

## Side effects

- The I flag is set.

<b>SEI : Set Interrupt Disable Flag</b>		4510			
$I \leftarrow 1$		N Z I C D V E			
		. . . . .			
<b>Addressing Mode</b>		<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied		SEI	78	1	1 <sup>s</sup>

<sup>s</sup> Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## SMB0

This instruction sets bit zero of the indicated address. No flags are modified, regardless of the result.

<b>SMB0 : Set Bit 0 in Base Page</b>		4510			
$M(0) \leftarrow 1$		N Z I C D V E			
		. . . . .			
<b>Addressing Mode</b>		<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>	
base-page		SMB0 \$nn	87	2	4 <sup>r</sup>

<sup>r</sup> Add one cycle if clock speed is at 40 MHz.

## SMB1

This instruction sets bit 1 of the indicated address. No flags are modified, regardless of the result.

<b>SMB1 : Set Bit 1 in Base Page</b>		4510			
$M(1) \leftarrow 1$		N Z I C D V E			
		. . . . .			
<b>Addressing Mode</b>		<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>	
base-page		SMB1 \$nn	97	2	4 <sup>r</sup>

<sup>r</sup> Add one cycle if clock speed is at 40 MHz.

## SMB2

This instruction sets bit 2 of the indicated address. No flags are modified, regardless of the result.

<b>SMB2 : Set Bit 2 in Base Page</b>		<b>4510</b>
$M(2) \leftarrow 1$		
		<b>N Z I C D V E</b>
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
base-page	SMB2 \$nn	A7
		2
		4
		<i>r</i>

*r* Add one cycle if clock speed is at 40 MHz.

## SMB3

This instruction sets bit 3 of the indicated address. No flags are modified, regardless of the result.

<b>SMB3 : Set Bit 3 in Base Page</b>		<b>4510</b>
$M(3) \leftarrow 1$		
		<b>N Z I C D V E</b>
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
base-page	SMB3 \$nn	B7
		2
		4
		<i>r</i>

*r* Add one cycle if clock speed is at 40 MHz.

## SMB4

This instruction sets bit 4 of the indicated address. No flags are modified, regardless of the result.

<b>SMB4 : Set Bit 4 in Base Page</b>		<b>4510</b>
$M(4) \leftarrow 1$		
		<b>N Z I C D V E</b>
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
base-page	SMB4 \$nn	C7
		2
		4
		<i>r</i>

*r* Add one cycle if clock speed is at 40 MHz.

## SMB5

This instruction sets bit 5 of the indicated address. No flags are modified, regardless of the result.

<b>SMB5 : Set Bit 5 in Base Page</b>		<b>4510</b>
$M(5) \leftarrow 1$		
		<b>N Z I C D V E</b>
		.....
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
base-page	SMB5 \$nn	D7
		2
		4
<i>r</i> Add one cycle if clock speed is at 40 MHz.		<i>r</i>

## SMB6

This instruction sets bit 6 of the indicated address. No flags are modified, regardless of the result.

<b>SMB6 : Set Bit 6 in Base Page</b>		<b>4510</b>
$M(6) \leftarrow 1$		
		<b>N Z I C D V E</b>
		.....
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
base-page	SMB6 \$nn	E7
		2
		4
<i>r</i> Add one cycle if clock speed is at 40 MHz.		<i>r</i>

## SMB7

This instruction sets bit 7 of the indicated address. No flags are modified, regardless of the result.

<b>SMB7 : Set Bit 7 in Base Page</b>		<b>4510</b>
$M(7) \leftarrow 1$		
		<b>N Z I C D V E</b>
		.....
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
base-page	SMB7 \$nn	F7
		2
		4
<i>r</i> Add one cycle if clock speed is at 40 MHz.		<i>r</i>

# STA

This instruction stores the contents of the Accumulator Register into the indicated location.

STA : Store Accumulator		4510						
M ← A		N	Z	I	C	D	V	E
		.	.	.	.	.	.	.
Addressing Mode	Assembly	Code	Bytes					
(indirect,X)	STA (\$nn,X)	81	2	5				<i>p</i>
(indirect,SP),Y	STA (\$nn,SP),Y	82	2	6				<i>p</i>
base-page	STA \$nn	85	2	3				
absolute	STA \$nnnn	8D	3	4				
(indirect),Y	STA (\$nn),Y	91	2	5				<i>p</i>
(indirect),Z	STA (\$nn),Z	92	2	5				<i>p</i>
base-page,X	STA \$nn,X	95	2	3				<i>p</i>
absolute,Y	STA \$nnnn,Y	99	3	4				<i>p</i>
absolute,X	STA \$nnnn,X	9D	3	4				<i>p</i>

*p* Add one cycle if indexing crosses a page boundary.

# STX

This instruction stores the contents of the X Register into the indicated location.

STX : Store X Register		4510						
M ← X		N	Z	I	C	D	V	E
		.	.	.	.	.	.	.
Addressing Mode	Assembly	Code	Bytes					
base-page	STX \$nn	86	2	3				
absolute	STX \$nnnn	8E	3	4				
base-page,Y	STX \$nn,Y	96	2	3				<i>p</i>
absolute,Y	STX \$nnnn,Y	9B	3	4				<i>p</i>

*p* Add one cycle if indexing crosses a page boundary.

# STY

This instruction stores the contents of the Y Register into the indicated location.

STY : Store Y Register		4510						
M ← Y		N	Z	I	C	D	V	E
Addressing Mode	Assembly	Code	Bytes					
base-page	STY \$nn	84	2	3				
absolute,X	STY \$nnnn,X	8B	3	4				p
absolute	STY \$nnnn	8C	3	4				
base-page,X	STY \$nn,X	94	2	3				p

*p* Add one cycle if indexing crosses a page boundary.

# STZ

This instruction stores the contents of the Z Register into the indicated location.

STZ : Store Z Register		4510						
M ← Z		N	Z	I	C	D	V	E
Addressing Mode	Assembly	Code	Bytes					
base-page	STZ \$nn	64	2	3				
base-page,X	STZ \$nn,X	74	2	3				p
absolute	STZ \$nnnn	9C	3	4				
absolute,X	STZ \$nnnn,X	9E	3	4				p

*p* Add one cycle if indexing crosses a page boundary.

# TAB

This instruction sets the Base Page register to the contents of the Accumulator Register. This allows the relocation of the 6502's Zero-Page into any page of memory.

**TAB : Transfer Accumulator into Base Page Register      4510**B  $\leftarrow$  A**N Z I C D V E**

+ + . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	TAB	5B	1	1	<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.**TAX**

This instruction loads the X Register with the contents of the Accumulator Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TAX : Transfer Accumulator Register into the X Register      4510**X  $\leftarrow$  A**N Z I C D V E**

+ + . . . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	TAX	AA	1	1	<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.**TAY**

This instruction loads the Y Register with the contents of the Accumulator Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TAY : Transfer Accumulator Register into the Y Register 4510**

Y ← A

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

**Addressing Mode**

implied

**Assembly Code**

TAY

**Bytes**

A8

**Cycles**

1

1

*s**s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## TAZ

This instruction loads the Z Register with the contents of the Accumulator Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TAZ : Transfer Accumulator Register into the Z Register 4510**

Z ← A

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

**Addressing Mode**

implied

**Assembly Code**

TAZ

**Bytes**

4B

**Cycles**

1

1

*s**s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## TBA

This instruction loads the Accumulator Register with the contents of the Base Page Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TBA : Transfer Base Page Register into the Accumulator 4510**A  $\leftarrow$  B

N	Z	I	C	D	V	E
+	+	.	.	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
implied	TBA	7B	1	1	<sup>s</sup>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

**TRB**

This instruction sets performs a binary AND of the negation of the Accumulator Register and the indicated memory location, storing the result there. That is, any bits set in the Accumulator Register will be reset in the indicated memory location.

It also performs a test for any bits in common between the accumulator and indicated memory location. This can be used to construct simple shared-memory multi-processor systems, by providing an atomic means of setting a semaphore or acquiring a lock.

**Side effects**

- The Z flag will be set if the binary AND of the Accumulator Register and contents of the indicated memory location prior are zero, prior to the execution of the instruction.

**TRB : Test and Reset Bit****4510**M  $\leftarrow$  M AND (*NOT A*)

N	Z	I	C	D	V	E
.	+	.	.	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page	TRB \$nn	14	2	4	<sup>mr</sup>
absolute	TRB \$nnnn	1C	3	5	<sup>mr</sup>

*m* Subtract non-bus cycles when at 40MHz.

*r* Add one cycle if clock speed is at 40 MHz.

# TSB

This instruction sets performs a binary OR of the Accumulator Register and the indicated memory location, storing the result there. That is, any bits set in the Accumulator Register will be set in the indicated memory location.

It also performs a test for any bits in common between the accumulator and indicated memory location. This can be used to construct simple shared-memory multi-processor systems, by providing an atomic means of setting a semaphore or acquiring a lock.

## Side effects

- The Z flag will be set if the binary AND of the Accumulator Register and contents of the indicated memory location prior are zero, prior to the execution of the instruction.

TSB : Test and Set Bit		4510						
$M \leftarrow M \text{ OR } A$		N	Z	I	C	D	V	E
base-page	TSB \$nn	04	2	4				<i>mr</i>
absolute	TSB \$nnnn	0C	3	5				<i>mr</i>

*m* Subtract non-bus cycles when at 40MHz.

*r* Add one cycle if clock speed is at 40 MHz.

# TSX

This instruction loads the X Register with the contents of the Stack Pointer High (SPL) Register.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TSX : Transfer Stack Pointer High Register into the X Register 4510**X  $\leftarrow$  SPH

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	TSX	BA	1	1	<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## TSY

This instruction loads the Y Register with the contents of the Stack Pointer High (SPH) Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TSY : Transfer Stack Pointer High Register into the Y Register 4510**Y  $\leftarrow$  SPH

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	TSY	0B	1	1	<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## TXA

This instruction loads the Accumulator Register with the contents of the X Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TXA : Transfer X Register into the Accumulator Register 4510**A  $\leftarrow$  X

N	Z	I	C	D	V	E
+	+	.	.	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
implied	TXA	8A	1	1	s

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

**TXS**

This instruction sets the low byte of the Stack Pointer (SPL) register to the contents of the X Register.

**TXS : Transfer X Register into Stack Pointer Low Register 4510**SPL  $\leftarrow$  X

N	Z	I	C	D	V	E
.	.	.	.	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
implied	TXS	9A	1	1	s

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

**TYA**

This instruction loads the Accumulator Register with the contents of the Y Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TYA : Transfer Y Register into the Accumulator Register 4510**A  $\leftarrow$  Y

N	Z	I	C	D	V	E
+	+	.	.	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
implied	TYA	98	1	1	s

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## TYS

This instruction sets the high byte of the Stack Pointer (SPH) register to the contents of the Y Register. This allows changing the memory page where the stack is located (if the Extended Stack Disable Flag (E) is set), or else allows setting the current Stack Pointer to any page in memory, if the Extended Stack Disable Flag (E) is clear.

<b>TYS : Transfer Y Register into Stack Pointer High Register    4510</b>					
SPH ← Y					
N Z I C D V E					
. . . . .					
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	TYS	2B	1	1	<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## TZA

This instruction loads the Accumulator Register with the contents of the Z Register.

### Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>TZA : Transfer Z Register into the Accumulator Register    4510</b>					
A ← Z					
N Z I C D V E					
+ + . . .					
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
implied	TZA	6B	1	1	<i>s</i>

*s* Instruction requires 2 cycles when CPU is run at 1MHz or 2MHz.

## 6502 INSTRUCTION SET

NOTE: The mechanisms for switching from 4510 to 6502 CPU personality have yet to be finalised.

NOTE: Not all 6502 illegal opcodes are currently implemented.

## Opcode Map

\$x0	\$x1	\$x2	\$x3	\$x4	\$x5	\$x6	\$x7	\$x8	\$x9	\$xA	\$xB	\$xC	\$xD	\$xE	\$xF
\$0x	BRK	ORA	KIL	SLO	NOP	ORA	ASL	SLO	PHP	ORA	ASL	ANC	NOP	ORA	ASL
\$1x	BPL	ORA	KIL	SLO	NOP	ORA	ASL	SLO	CLC	ORA	NOP	SLO	NOP	ORA	ASL
\$2x	JSR	AND	KIL	RLA	BIT	AND	ROL	RLA	PLP	AND	ROL	ANC	BIT	AND	ROL
\$3x	BMI	AND	KIL	RLA	NOP	AND	ROL	RLA	SEC	AND	NOP	RLA	NOP	AND	ROL
\$4x	RTI	EOR	KIL	SRE	NOP	EOR	LSR	SRE	PHA	EOR	LSR	ALR	JMP	EOR	LSR
\$5x	BVC	EOR	KIL	SRE	NOP	EOR	LSR	SRE	CLI	EOR	NOP	SRE	NOP	EOR	LSR
\$6x	RTS	ADC	KIL	RRA	NOP	ADC	ROR	RRA	PLA	ADC	ROR	ARR	JMP	ADC	ROR
\$7x	BVS	ADC	KIL	RRA	NOP	ADC	ROR	RRA	SEI	ADC	NOP	RRA	NOP	ADC	ROR
\$8x	NOP	STA	NOP	SAX	STY	STA	STX	SAX	DEY	NOP	TXA	XAA	STY	STA	STX
\$9x	BCC	STA	KIL	SHA	STY	STA	STX	SAX	TYA	STA	TXS	TAS	SHY	STA	SHX
\$Ax	LDY	LDA	LDX	LAX	LDY	LDA	LDX	LAX	TAY	LDA	TAX	LAX	LDY	LDA	LDX
\$Bx	BCS	LDA	KIL	LAX	LDY	LDA	LDX	LAX	CIV	LDA	TSX	LAS	LDY	LDA	LDX
\$Cx	CPY	CMP	NOP	DCP	CPY	CMP	DEC	DCP	INY	CMP	DEX	SBX	CPY	CMP	DEC
\$Dx	BNE	CMP	KIL	DCP	NOP	CMP	DEC	DCP	CLD	CMP	NOP	DCP	NOP	CMP	DEC
\$Ex	CPX	SBC	NOP	ISC	CPX	SBC	INC	ISC	INX	SBC	NOP	SBC	CPX	SBC	INC
\$Fx	BEQ	SBC	KIL	ISC	NOP	SBC	INC	ISC	SED	SBC	NOP	ISC	INC	ISC	ISC

# Instruction Timing

The following table summarises the base instruction timing for 6502 mode. Please also read the information for 4510 mode, as it discusses a number of important factors that affect these figures.

	\$x0	\$x1	\$x2	\$x3	\$x4	\$x5	\$x6	\$x7	\$x8	\$x9	\$xA	\$xB	\$xC	\$xD	\$xE	\$xF
\$0x	7	6	9	8	3	5	5	3	2	2	2	4	4	6	6	6
\$1x	2 <sup>b</sup>	5 <sup>p</sup>	9	8	4	4 <sup>p</sup>	6	6	2	4 <sup>p</sup>	2	7	4 <sup>p</sup>	4 <sup>p</sup>	7	7
\$2x	6	6	9	8	3	3	5	5	4	2	2	2	4	4	6	6
\$3x	2 <sup>b</sup>	5 <sup>p</sup>	9	8	4	4	6	6	2	4 <sup>p</sup>	2	7	4 <sup>p</sup>	4 <sup>p</sup>	7	7
\$4x	6	9	8	3	3	5	5	3	2	2	2	3	4	6	6	6
\$5x	2 <sup>b</sup>	5 <sup>p</sup>	9	8	4	4	6	6	2	4 <sup>p</sup>	2	7	4 <sup>p</sup>	4 <sup>p</sup>	7	7
\$6x	6	6	9	8	3	3	5	5	4	2	2	2	5	4	6	6
\$7x	2 <sup>b</sup>	5 <sup>p</sup>	9	8	4	4	6	6	2	4 <sup>p</sup>	2	7	4 <sup>p</sup>	4 <sup>p</sup>	7	7
\$8x	2	6	2	6	3	3	3	3	2	2	2	4	4	4	4	4
\$9x	2 <sup>b</sup>	6	9	6	4	4	4	4	2	5	2	5	5	5	5	5
\$Ax	2	6	2	6	3	3	3	3	2	2	2	4	4	4	4	4
\$Bx	2 <sup>b</sup>	5 <sup>p</sup>	9	5 <sup>p</sup>	4	4	4	4	2	4 <sup>p</sup>	2	4 <sup>p</sup>				
\$Cx	2	6	2	8	3	3	5	5	2	2	2	4	4	6	6	6
\$Dx	2 <sup>b</sup>	5 <sup>p</sup>	9	8	4	4	6	6	2	4 <sup>p</sup>	2	7	4 <sup>p</sup>	4 <sup>p</sup>	7	7
\$Ex	2	6	2	8	3	3	5	5	2	2	2	4	4	6	6	6
\$Fx	2 <sup>b</sup>	5 <sup>p</sup>	9	8	4	4	6	6	2	4 <sup>p</sup>	2	7	4 <sup>p</sup>	4 <sup>p</sup>	7	7

<sup>b</sup> Add one cycle if branch crosses a page boundary.

<sup>p</sup> Add one cycle if indexing crosses a page boundary.

## **Addressing Mode Table**

\$x.0	\$x.1	\$x.2	\$x.3	\$x.4	\$x.5	\$x.6	\$x.7	\$x.8	\$x.9	\$x.A	\$x.B	\$x.C	\$x.D	\$x.E	\$x.F
\$0x	\$0n,X	\$0n,Y	\$0n,X	\$0n,Y	\$0n,X	\$0n,Y	\$0n,X	\$0n,Y	\$0n,X	\$#0n	A	\$#0n	\$0nnn	\$0nnn	\$0nnn
\$1x	\$1n,Y	\$1n,X	\$1n,Y	\$1n,X	\$1n,Y	\$1n,X	\$1n,Y	\$1n,X	\$1n,Y	\$#1n	A	\$#1n	\$0nnn	\$0nnn	\$0nnn
\$2x	\$2n,NN	\$2n,X	\$2n,NN	\$2n,X	\$2n,NN	\$2n,X	\$2n,NN	\$2n,X	\$2n,NN	\$#2n	A	\$#2n	\$0nnn	\$0nnn	\$0nnn
\$3x	\$3n,Y	\$3n,X	\$3n,Y	\$3n,X	\$3n,Y	\$3n,X	\$3n,Y	\$3n,X	\$3n,Y	\$#3n	A	\$#3n	\$0nnn	\$0nnn	\$0nnn
\$4x	\$4n,X	\$4n,Y	\$4n,X	\$4n,Y	\$4n,X	\$4n,Y	\$4n,X	\$4n,Y	\$4n,X	\$#4n	A	\$#4n	\$0nnn	\$0nnn	\$0nnn
\$5x	\$5n,Y	\$5n,X	\$5n,Y	\$5n,X	\$5n,Y	\$5n,X	\$5n,Y	\$5n,X	\$5n,Y	\$#5n	A	\$#5n	\$0nnn	\$0nnn	\$0nnn
\$6x	\$6n,X	\$6n,Y	\$6n,X	\$6n,Y	\$6n,X	\$6n,Y	\$6n,X	\$6n,Y	\$6n,X	\$#6n	A	\$#6n	\$0nnn	\$0nnn	\$0nnn
\$7x	\$7n,Y	\$7n,X	\$7n,Y	\$7n,X	\$7n,Y	\$7n,X	\$7n,Y	\$7n,X	\$7n,Y	\$#7n	A	\$#7n	\$0nnn	\$0nnn	\$0nnn
\$8x	\$8n,NN	\$8n,X	\$8n,NN	\$8n,X	\$8n,NN	\$8n,X	\$8n,NN	\$8n,X	\$8n,NN	\$#8n	A	\$#8n	\$0nnn	\$0nnn	\$0nnn
\$9x	\$9n,Y	\$9n,X	\$9n,Y	\$9n,X	\$9n,Y	\$9n,X	\$9n,Y	\$9n,X	\$9n,Y	\$#9n	A	\$#9n	\$0nnn	\$0nnn	\$0nnn
\$Ax	\$#0nn	\$#1nn	\$#2nn	\$#3nn	\$#4nn	\$#5nn	\$#6nn	\$#7nn	\$#8nn	\$#9nn	\$#A	\$#A	\$#B	\$#C	\$#D
\$Bx	\$#0nn	\$#1nn	\$#2nn	\$#3nn	\$#4nn	\$#5nn	\$#6nn	\$#7nn	\$#8nn	\$#9nn	\$#A	\$#A	\$#B	\$#C	\$#D
\$Cx	\$#0nn	\$#1nn	\$#2nn	\$#3nn	\$#4nn	\$#5nn	\$#6nn	\$#7nn	\$#8nn	\$#9nn	\$#A	\$#A	\$#B	\$#C	\$#D
\$Dx	\$#0nn	\$#1nn	\$#2nn	\$#3nn	\$#4nn	\$#5nn	\$#6nn	\$#7nn	\$#8nn	\$#9nn	\$#A	\$#A	\$#B	\$#C	\$#D
\$Ex	\$#0nn	\$#1nn	\$#2nn	\$#3nn	\$#4nn	\$#5nn	\$#6nn	\$#7nn	\$#8nn	\$#9nn	\$#A	\$#A	\$#B	\$#C	\$#D
\$Fx	\$#0nn	\$#1nn	\$#2nn	\$#3nn	\$#4nn	\$#5nn	\$#6nn	\$#7nn	\$#8nn	\$#9nn	\$#A	\$#A	\$#B	\$#C	\$#D

# Official And Unintended Instructions

The 6502 opcode matrix has a size of  $16 \times 16 = 256$  possible opcodes. Those, that are officially documented, form the set of the [legal](#) instructions. All instructions of this legal set are headed by a blue coloured mnemonic.

The remaining opcodes form the set of the [unintended](#) instructions (sometimes called "illegal" instructions). For the sake of completeness these are documented too. All instructions of the unintended set are headed by a red coloured mnemonic.

The unintended instructions are implemented in the 6502 mode, but are not guaranteed to produce exactly the same results as on other CPU's of the 65xx family. Many of these instructions are known to be unstable, even running on old hardware.

## ADC

This instruction adds the argument to the contents of the Accumulator Register and the Carry Flag. If the D flag is set, then the addition is performed using Binary Coded Decimal.

### Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, else it will be cleared. If the flag is set, this indicates that a signed overflow has occurred.
- The C flag will be set if the unsigned result is  $>255$ , or  $>99$  if the D flag is set.

**ADC : Add with carry****6502** $A \leftarrow A + M + C$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	+	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	ADC (\$nn,X)	61	2	6	
zero-page	ADC \$nn	65	2	3	
immediate	ADC #\$nn	69	2	2	
absolute	ADC \$nnnn	6D	3	4	
(indirect),Y	ADC (\$nn),Y	71	2	5	<i>p</i>
zero-page,X	ADC \$nn,X	75	2	4	
absolute,Y	ADC \$nnnn,Y	79	3	4	<i>p</i>
absolute,X	ADC \$nnnn,X	7D	3	4	<i>p</i>

*p* Add one cycle if indexing crosses a page boundary.

**ALR**

This instruction shifts the Accumulator one bit right after performing a binary AND of the Accumulator and the immediate mode argument. Bit 7 will be set to zero, and the bit 0 will be shifted out into the Carry Flag

**Side effects**

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 0 of the value was set, prior to being shifted.

**ALR : Binary AND and Logical Shift Right****6502** $A \leftarrow (A \text{ AND Value}) >> 1, C \leftarrow A(0)$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	+	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
immediate	ALR #\$nn	4B	2	2

**ANC**

This instruction performs a binary AND operation of the argument with the accumulator, and stores the result in the accumulator. Only bits that were

already set in the accumulator, and that are set in the argument will be set in the accumulator on completion. Unlike the AND instruction, the Carry Flag is set as though the result were shifted left one bit. That is, the Carry Flag is set in the same way as the Negative Flag.

### **Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The C flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>ANC : Binary AND, and Set Carry</b>		<b>6502</b>
A $\leftarrow$ A AND M, C $\leftarrow$ A7 AND M7		
		N Z I C D V E
		+ + . . . . .
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
immediate	ANC #\$nn	0B
immediate	ANC #\$nn	2B

## **AND**

This instruction performs a binary AND operation of the argument with the accumulator, and stores the result in the accumulator. Only bits that were already set in the accumulator, and that are set in the argument will be set in the accumulator on completion.

### **Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**AND : Binary AND****6502** $A \leftarrow A \text{ AND } M$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	AND (\$nn,X)	21	2	6	
zero-page	AND \$nn	25	2	3	
immediate	AND #\$nn	29	2	2	
absolute	AND \$nnnn	2D	3	4	
(indirect),Y	AND (\$nn),Y	31	2	5	<i>p</i>
zero-page,X	AND \$nn,X	35	2	4	
absolute,Y	AND \$nnnn,Y	39	3	4	<i>p</i>
absolute,X	AND \$nnnn,X	3D	3	4	<i>p</i>

*p* Add one cycle if indexing crosses a page boundary.

**ARR**

This instruction shifts the Accumulator one bit right after performing a binary AND of the Accumulator and the immediate mode argument. Bit 7 is exchanged with the carry.

**Side effects**

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The V flag will be apparently be affected in some way.
- The C flag will be set if bit 7 of the value was set, prior to being shifted.

**ARR : Binary AND and Rotate Right****6502** $A \leftarrow (A \text{ AND Value}) >> 1, C \leftarrow A(7)$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	+	+	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
immediate	ARR #\$nn	6B	2	2

# ASL

This instruction shifts either the Accumulator or contents of the provided memory location one bit left. Bit 0 will be set to zero, and the bit 7 will be shifted out into the Carry Flag

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 7 of the value was set, prior to being shifted.

ASL : Arithmetic Shift Left		6502						
A $\leftarrow$ A<<1 or M $\leftarrow$ M<<1		N	Z	I	C	D	V	E
		+	+	.	+	.	.	.
Addressing Mode	Assembly	Code	Bytes					
zero-page	ASL \$nn	06	2					
accumulator	ASL A	0A	1					
absolute	ASL \$nnnn	0E	3					
zero-page,X	ASL \$nn,X	16	2					
absolute,X	ASL \$nnnn,X	1E	3					

# BCC

This instruction branches to the indicated address if the Carry Flag is clear.

BCC : Branch on Carry Flag Clear		6502						
PC $\leftarrow$ PC + R8		N	Z	I	C	D	V	E
		.	.	.	.	.	.	.
Addressing Mode	Assembly	Code	Bytes					
relative	BCC \$rr	90	2					<sup>b</sup>

<sup>b</sup> Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

# BCS

This instruction branches to the indicated address if the Carry Flag is set.

<b>BCS : Branch on Carry Flag Set</b>		<b>6502</b>	
PC ← PC + R8		N Z I C D V E	
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>
relative	BCS \$rr	B0	2      2 <sup>b</sup>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

# BEQ

This instruction branches to the indicated address if the Zero Flag is set.

<b>BEQ : Branch on Zero Flag Set</b>		<b>6502</b>	
PC ← PC + R8		N Z I C D V E	
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>
relative	BEQ \$rr	F0	2      2 <sup>b</sup>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

# BIT

This instruction is used to test the bits stored in a memory location. Bits 6 and 7 of the memory location's contents are directly copied into the Overflow Flag and Negative Flag. The Zero Flag is set or cleared based on the result of performing the binary AND of the Accumulator Register and the contents of the indicated memory location.

## Side effects

- The N flag will be set if the bit 7 of the memory location is set, else it will be cleared.

- The V flag will be set if the bit 6 of the memory location is set, else it will be cleared.
- The Z flag will be set if the result of A AND M is zero, else it will be cleared.

<b>BIT : Perform Bit Test</b>		<b>6502</b>
N $\leftarrow$ M(7), V $\leftarrow$ M(6), Z $\leftarrow$ A AND M		
		N Z I C D V E + + . . . + .
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>
zero-page	BIT \$nn	24
absolute	BIT \$nnnn	2C
	<b>Bytes</b>	<b>Cycles</b>
	2	3
	3	4

## BMI

This instruction branches to the indicated address if the Negative Flag is set.

<b>BMI : Branch on Negative Flag Set</b>		<b>6502</b>
PC $\leftarrow$ PC + R8		
		N Z I C D V E . . . . .
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>
relative	BMI \$rr	30
	<b>Bytes</b>	<b>Cycles</b>
	2	2 <sup>b</sup>

<sup>b</sup> Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

## BNE

This instruction branches to the indicated address if the Zero Flag is clear.

<b>BNE : Branch on Zero Flag Clear</b>		<b>6502</b>
PC $\leftarrow$ PC + R8		
		N Z I C D V E . . . . .
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>
relative	BNE \$rr	D0
	<b>Bytes</b>	<b>Cycles</b>
	2	2 <sup>b</sup>

<sup>b</sup> Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

## BPL

This instruction branches to the indicated address if the Negative Flag is clear.

BPL : Branch on Negative Flag Clear		6502
PC ← PC + R8		N Z I C D V E
		.....
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>
relative	BPL \$rr	10 2 2 <sup>b</sup>

<sup>b</sup> Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

## BRK

The break command causes the microprocessor to go through an interrupt sequence under program control. The address of the BRK instruction + 2 is pushed to the stack along with the status register with the Break flag set. This allows the interrupt service routine to distinguish between IRQ events and BRK events. For example:

```
PLA          ; load status
PHA          ; restore stack
AND #$10    ; mask break flag
BNE DO_BREAK ; -> it was a BRK
...          ; else continue with IRQ server
```

Cite from: MCS6500 Microcomputer Family Programming Manual, January 1976, Second Edition, MOS Technology Inc., Page 144:

"The BRK is a single byte instruction and its addressing mode is Implied."

There are debates, that BRK could be seen as a two byte instruction with the addressing mode immediate, where the operand byte is discarded. The byte following the BRK could then be used as a call argument for the break handler. Commodore however used the BRK, as stated in the manual, as a single byte instruction, which breaks into the ML monitor, if present. These builtin monitors decremented the stacked PC, so that it could be used to return or jump directly to the code byte after the BRK.

<b>BRK : Break to Interrupt</b>	<b>6502</b>			
PC $\leftarrow$ (\$FFFF)				
<b>N Z I C D V E</b>				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	BRK	00	1	7

## BVC

This instruction branches to the indicated address if the Overflow (V) Flag is clear.

<b>BVC : Branch on Overflow Flag Clear</b>	<b>6502</b>			
PC $\leftarrow$ PC + R8				
<b>N Z I C D V E</b>				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
relative	BVC \$rr	50	2	2 <sup>b</sup>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

## BVS

This instruction branches to the indicated address if the Overflow (V) Flag is set.

<b>BVS : Branch on Overflow Flag Set</b>	<b>6502</b>			
PC $\leftarrow$ PC + R8				
<b>N Z I C D V E</b>				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
relative	BVS \$rr	70	2	2 <sup>b</sup>

*b* Add one cycle if branch is taken.

Add one more cycle if branch taken crosses a page boundary.

## CLC

This instruction clears the Carry Flag.

## Side effects

- The C flag is cleared.

<b>CLC : Clear Carry Flag</b>		<b>6502</b>	
C ← 0		N Z I C D V E	
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	CLC	18	1 2

# CLD

This instruction clears the Decimal Flag. Arithmetic operations will use normal binary arithmetic, instead of Binary-Coded Decimal (BCD).

## Side effects

- The D flag is cleared.

<b>CLD : Clear Decimal Flag</b>		<b>6502</b>	
D ← 0		N Z I C D V E	
<b>Addressing Mode</b>	<b>Assembly Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	CLD	D8	1 2

# CLI

This instruction clears the Interrupt Disable Flag. Interrupts will now be able to occur.

## Side effects

- The I flag is cleared.

<b>CLI : Clear Interrupt Disable Flag</b>	<b>6502</b>			
I ← 0				
N Z I C D V E				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	CLI	58	1	2

## CLV

This instruction clears the Overflow Flag.

### Side effects

- The V flag is cleared.

<b>CLV : Clear Overflow Flag</b>	<b>6502</b>			
V ← 0				
N Z I C D V E				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	CLV	B8	1	2

## CMP

This instruction performs A – M, and sets the processor flags accordingly, but does not modify the contents of the Accumulator Register.

### Side effects

- The N flag will be set if the result of A – M is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of A – M is zero or positive, i.e., if A is not less than M, else it will be cleared.
- The Z flag will be set if the result of A – M is zero, else it will be cleared.

**CMP : Compare Accumulator****6502**

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	CMP (\$nn,X)	C1	2	6	
zero-page	CMP \$nn	C5	2	3	
immediate	CMP #\$nn	C9	2	2	
absolute	CMP \$nnnn	CD	3	4	
(indirect),Y	CMP (\$nn),Y	D1	2	5	<i>p</i>
zero-page,X	CMP \$nn,X	D5	2	4	
absolute,Y	CMP \$nnnn,Y	D9	3	4	<i>p</i>
absolute,X	CMP \$nnnn,X	DD	3	4	<i>p</i>

*p* Add one cycle if indexing crosses a page boundary.

**CPX**

This instruction performs  $X - M$ , and sets the processor flags accordingly, but does not modify the contents of the Accumulator Register.

**Side effects**

- The N flag will be set if the result of  $X - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of  $X - M$  is zero or positive, i.e., if X is not less than M, else it will be cleared.
- The Z flag will be set if the result of  $X - M$  is zero, else it will be cleared.

**CPX : Compare X Register****6502**

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
immediate	CPX #\$nn	E0	2	2
zero-page	CPX \$nn	E4	2	3
absolute	CPX \$nnnn	EC	3	4

# CPY

This instruction performs  $Y - M$ , and sets the processor flags accordingly, but does not modify the contents of the Accumulator Register.

## Side effects

- The N flag will be set if the result of  $Y - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of  $Y - M$  is zero or positive, i.e., if  $Y$  is not less than  $M$ , else it will be cleared.
- The Z flag will be set if the result of  $Y - M$  is zero, else it will be cleared.

CPY : Compare Y Register		6502						
		N	Z	I	C	D	V	E
		+	+	.	+	.	.	.
Addressing Mode	Assembly	Code	Bytes	Bytes	Cycles			
immediate	CPY #\$nn	C0	2	2				
zero-page	CPY \$nn	C4	2	3				
absolute	CPY \$nnnn	CC	3	4				

# DCP

This instruction decrements the contents of the indicated memory location, and then performs  $A - M$ , and sets the processor flags accordingly, but does not modify the contents of the Accumulator Register.

## Side effects

- The N flag will be set if the result of  $A - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of  $A - M$  is zero or positive, i.e., if  $A$  is not less than  $M$ , else it will be cleared.
- The Z flag will be set if the result of  $A - M$  is zero, else it will be cleared.

**DCP : Decrement and Compare Accumulator** 6502 $M \leftarrow M - 1, A \leftarrow M$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
(indirect,X)	DCP (\$nn,X)	C3	2	8
zero-page	DCP \$nn	C7	2	5
absolute	DCP \$nnnn	CF	3	6
(indirect),Y	DCP (\$nn),Y	D3	2	8
zero-page,X	DCP \$nn,X	D7	2	6
absolute,Y	DCP \$nnnn,Y	DB	3	7
absolute,X	DCP \$nnnn,X	DF	3	7

## DEC

This instruction decrements the Accumulator Register or indicated memory location.

### Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**DEC : Decrement Memory or Accumulator** 6502 $A \leftarrow A - 1 \text{ or } M \leftarrow M - 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
zero-page	DEC \$nn	C6	2	5
absolute	DEC \$nnnn	CE	3	6
zero-page,X	DEC \$nn,X	D6	2	6
absolute,X	DEC \$nnnn,X	DE	3	7

## DEX

This instruction decrements the X Register.

### **Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>DEX : Decrement X Register</b>		<b>6502</b>		
X $\leftarrow$ X - 1				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	DEX	CA	1	2

## **DEY**

This instruction decrements the Y Register.

### **Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>DEY : Decrement Y Register</b>		<b>6502</b>		
Y $\leftarrow$ Y - 1				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	DEY	88	1	2

## **EOR**

This instruction performs a binary XOR operation of the argument with the accumulator, and stores the result in the accumulator. Only bits that were already set in the accumulator, or that are set in the argument will be set in the accumulator on completion, but not both.

### **Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**EOR : Binary Exclusive OR****6502** $A \leftarrow A \text{ XOR } M$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	EOR (\$nn,X)	41	2	6	
zero-page	EOR \$nn	45	2	3	
immediate	EOR #\$nn	49	2	2	
absolute	EOR \$nnnn	4D	3	4	
(indirect),Y	EOR (\$nn),Y	51	2	5	<i>p</i>
zero-page,X	EOR \$nn,X	55	2	4	
absolute,Y	EOR \$nnnn,Y	59	3	4	<i>p</i>
absolute,X	EOR \$nnnn,X	5D	3	4	<i>p</i>

*p* Add one cycle if indexing crosses a page boundary.

**INC**

This instruction increments the Accumulator Register or indicated memory location.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**INC : Increment Memory or Accumulator****6502** $A \leftarrow A + 1 \text{ or } M \leftarrow M + 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
zero-page	INC \$nn	E6	2	5
absolute	INC \$nnnn	EE	3	6
zero-page,X	INC \$nn,X	F6	2	6
absolute,X	INC \$nnnn,X	FE	3	7

**INX**

This instruction increments the X Register, i.e., adds 1 to it.

### **Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>INX : Increment X Register</b>		<b>6502</b>
$X \leftarrow X + 1$		
	<b>N Z I C D V E</b>	
	+ + . . . . .	
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>
implied	INX	E8
<b>Bytes</b>		<b>Cycles</b>
		2

## **INY**

This instruction increments the Y Register, i.e., adds 1 to it.

### **Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

<b>INY : Increment Y Register</b>		<b>6502</b>
$Y \leftarrow Y + 1$		
	<b>N Z I C D V E</b>	
	+ + . . . . .	
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>
implied	INY	C8
<b>Bytes</b>		<b>Cycles</b>
		2

## **ISC**

This instruction increments the indicated memory location, and then performs  $A - M - 1 + C$ , and sets the processor flags accordingly. The result is stored in the Accumulator Register.

NOTE: This instruction is affected by the status of the Decimal Flag.

### **Side effects**

- The N flag will be set if the result of  $A - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.

- The C flag will be set if the result of A – M is zero or positive, i.e., if A is not less than M, else it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, else it will be cleared. If the flag is set, this indicates that a signed overflow has occurred.
- The Z flag will be set if the result of A – M is zero, else it will be cleared.

### **ISC : Increment Memory, Subtract With Carry**

**6502**

$M \leftarrow M + 1, A \leftarrow -M - 1 + C$

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	+	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
(indirect,X)	ISC (\$nn,X)	E3	2	8
zero-page	ISC \$nn	E7	2	5
absolute	ISC \$nnnn	EF	3	6
(indirect),Y	ISC (\$nn),Y	F3	2	8
zero-page,X	ISC \$nn,X	F7	2	6
absolute,Y	ISC \$nnnn,Y	FB	3	7
absolute,X	ISC \$nnnn,X	FF	3	7

## **JMP**

This instruction sets the Programme Counter (PC) Register to the address indicated by the instruction, causing execution to continue from that address.

### **JMP : Jump to Address**

**6502**

$PC \leftarrow M_2:M_1$

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
.	.	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
absolute	JMP \$nnnn	4C	3	3
indirect	JMP (\$nnnn)	6C	3	5

## **JSR**

This instruction saves the address of the instruction following the JSR instruction onto the stack, and then sets the Programme Counter (PC) Register to the address indicated by the instruction, causing execution to continue from

that address. Because the return address has been saved on the stack, the RTS instruction can be used to return from the called sub-routine and resume execution following the JSR instruction.

NOTE: This instruction actually pushes the address of the last byte of the JSR instruction onto the stack. The RTS instruction naturally is aware of this, and increments the address on popping it from the stack, before setting the Programme Counter (PC) register.

<b>JSR : Jump to Sub-Routine</b>		<b>6502</b>						
PC ← M2:M1, Stack ← PCH:PCL		N	Z	I	C	D	V	E
.	.	.	.	.	.	.	.	.
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>				
absolute	JSR \$nnnn	20	3	6				

## KIL

On a 6502, these instructions cause the processor to enter an infinite loop in their internal logic that can only be aborted by resetting the computer. On the 45GS02 these instructions cause Hypervisor Traps. Or rather, they will, once this functionality has been implemented. Thus they can be used to detect whether running on a 6502 or a 45GS02: If on a 6502 processor, the instruction will never return, while they will cause an exception on a 45GS02, likely causing the calling programme to be aborted or crash.

N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	KIL	02	1	9
implied	KIL	12	1	9
implied	KIL	22	1	9
implied	KIL	32	1	9
implied	KIL	42	1	9
implied	KIL	52	1	9
implied	KIL	62	1	9
implied	KIL	72	1	9
implied	KIL	92	1	9
implied	KIL	B2	1	9
implied	KIL	D2	1	9
implied	KIL	F2	1	9

## LAS

NOTE: This monstrosity of an instruction, aside from being devoid of any conceivable useful purpose is unstable on many 6502 processors and should therefore also be avoided for that reason, if you had not already been put off.

### Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- A feeling of hollow satisfaction, when you actually discover a useful purpose for this instruction. exactly how it works.

SP, A, X  $\leftarrow$  SP AND M

N Z I C D

V E

Addressing Mode	Assembly	Code	Bytes	Cycles
absolute,Y	LAS \$nnnn,Y	BB	3	4

p Add one cycle if indexing crosses a page boundary.

# LAX

This instruction loads both the Accumulator Register and X Register with the indicated value, or with the contents of the indicated location.

NOTE: The LAX instruction is known to be unstable on many 6502 processors, and should not be used. Non-immediate modes MAY be stable enough to be usable, but should generally be avoided.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

LAX : Load Accumulator and X Registers		6502						
A, X ← M		N	Z	I	C	D	V	E
		+	+	.	.	.	.	.
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>				
(indirect,X)	LAX (\$nn,X)	A3	2	6				
zero-page	LAX \$nn	A7	2	3				
immediate	LAX #\$nn	AB	2	2				
absolute	LAX \$nnnn	AF	3	4				
(indirect),Y	LAX (\$nn),Y	B3	2	5				p
zero-page,Y	LAX \$nn,Y	B7	2	4				
absolute,Y	LAX \$nnnn,Y	BF	3	4				p

*p* Add one cycle if indexing crosses a page boundary.

# LDA

This instruction loads the Accumulator Register with the indicated value, or with the contents of the indicated location.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**LDA : Load Accumulator****6502**A  $\leftarrow$  M

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	LDA (\$nn,X)	A1	2	6	
zero-page	LDA \$nn	A5	2	3	
immediate	LDA #\$nn	A9	2	2	
absolute	LDA \$nnnn	AD	3	4	
(indirect),Y	LDA (\$nn),Y	B1	2	5	<i>p</i>
zero-page,X	LDA \$nn,X	B5	2	4	
absolute,Y	LDA \$nnnn,Y	B9	3	4	<i>p</i>
absolute,X	LDA \$nnnn,X	BD	3	4	<i>p</i>

*p* Add one cycle if indexing crosses a page boundary.

**LDX**

This instruction loads the X Register with the indicated value, or with the contents of the indicated location.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**LDX : Load X Register****6502**X  $\leftarrow$  M

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
immediate	LDX #\$nn	A2	2	2	
zero-page	LDX \$nn	A6	2	3	
absolute	LDX \$nnnn	AE	3	4	
zero-page,Y	LDX \$nn,Y	B6	2	4	
absolute,Y	LDX \$nnnn,Y	BE	3	4	<i>p</i>

*p* Add one cycle if indexing crosses a page boundary.

# LDY

This instruction loads the Y Register with the indicated value, or with the contents of the indicated location.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

LDY : Load Y Register		6502						
Y ← M		N	Z	I	C	D	V	E
		+	+	.	.	.	.	.
Addressing Mode	Assembly	Code	Bytes					Cycles
immediate	LDY #\$nn	A0	2					2
zero-page	LDY \$nn	A4	2					3
absolute	LDY \$nnnn	AC	3					4
zero-page,X	LDY \$nn,X	B4	2					4
absolute,X	LDY \$nnnn,X	BC	3					p

*p* Add one cycle if indexing crosses a page boundary.

# LSR

This instruction shifts either the Accumulator or contents of the provided memory location one bit right. Bit 7 will be set to zero, and the bit 0 will be shifted out into the Carry Flag

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 0 of the value was set, prior to being shifted.

**LSR : Logical Shift Right****6502** $A \leftarrow A >> 1, C \leftarrow A(0)$  or  $M \leftarrow M >> 1$ **N Z I C D V E**  
+ + . + . .

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
zero-page	LSR \$nn	46	2	5
accumulator	LSR A	4A	1	2
absolute	LSR \$nnnn	4E	3	6
zero-page,X	LSR \$nn,X	56	2	6
absolute,X	LSR \$nnnn,X	5E	3	7

**NOP**

These instructions act as null instructions: They perform the bus accesses as though they were real instructions, but then do nothing with the retrieved value. They can thus be used either as delay instructions, or to read from registers that have side-effects when read, without corrupting a register.

Only \$EA is an intended opcode for NOP on the 6502. All others are only available on NMOS versions of the processor, or the 45GS02 in 6502 mode.

N Z I C D V E

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
zero-page	NOP \$nn	04	2	3
absolute	NOP \$nnnn	0C	3	4
zero-page,X	NOP \$nn,X	14	2	4
implied	NOP	1A	1	2
absolute,X	NOP \$nnnn,X	1C	3	4
zero-page,X	NOP \$nn,X	34	2	4
implied	NOP	3A	1	2
absolute,X	NOP \$nnnn,X	3C	3	4
zero-page	NOP \$nn	44	2	3
zero-page,X	NOP \$nn,X	54	2	4
implied	NOP	5A	1	2
absolute,X	NOP \$nnnn,X	5C	3	4
zero-page	NOP \$nn	64	2	3
zero-page,X	NOP \$nn,X	74	2	4
implied	NOP	7A	1	2
absolute,X	NOP \$nnnn,X	7C	3	4
immediate	NOP #\$nn	80	2	2
immediate	NOP #\$nn	82	2	2
immediate	NOP #\$nn	89	2	2
immediate	NOP #\$nn	C2	2	2
zero-page,X	NOP \$nn,X	D4	2	4
implied	NOP	DA	1	2
absolute,X	NOP \$nnnn,X	DC	3	4
immediate	NOP #\$nn	E2	2	2
implied	NOP	EA	1	2
zero-page,X	NOP \$nn,X	F4	2	4
implied	NOP	FA	1	2
absolute,X	NOP \$nnnn,X	FC	3	4

*p* Add one cycle if indexing crosses a page boundary.

## ORA

This instruction performs a binary OR operation of the argument with the accumulator, and stores the result in the accumulator. Only bits that were already set in the accumulator, or that are set in the argument will be set in the accumulator on completion, or both.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

ORA : Decrement Memory or Accumulator					6502
					A $\leftarrow$ A + 1 or M $\leftarrow$ M + 1
					N Z I C D V E
					+ + . . .
Addressing Mode	Assembly	Code	Bytes	Cycles	
(indirect,X)	ORA (\$nn,X)	01	2	6	
zero-page	ORA \$nn	05	2	3	
immediate	ORA #\$nn	09	2	2	
absolute	ORA \$nnnn	0D	3	4	
(indirect),Y	ORA (\$nn),Y	11	2	5	<i>p</i>
zero-page,X	ORA \$nn,X	15	2	4	<i>p</i>
absolute,Y	ORA \$nnnn,Y	19	3	4	<i>p</i>
absolute,X	ORA \$nnnn,X	1D	3	4	<i>p</i>

*p* Add one cycle if indexing crosses a page boundary.

## PHA

This instruction pushes the contents of the Accumulator Register onto the stack, and decrements the value of the Stack Pointer by 1.

PHA : Push Accumulator Register onto the Stack					6502
STACK $\leftarrow$ A, SP $\leftarrow$ SP - 1					
					N Z I C D V E
					. . .
Addressing Mode	Assembly	Code	Bytes	Cycles	
implied	PHA	48	1	3	

## PHP

This instruction pushes the contents of the Processor Flags onto the stack, and decrements the value of the Stack Pointer by 1.

<b>PHP : Push Processor Flags onto the Stack</b>		<b>6502</b>		
STACK $\leftarrow$ P, SP $\leftarrow$ SP - 1				
N Z I C D V E				
+ + . . . .				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	PHP	08	1	3

## PLA

This instruction replaces the contents of the Accumulator Register with the top value from the stack, and increments the value of the Stack Pointer by 1.

<b>PLA : Pull Accumulator Register from the Stack</b>		<b>6502</b>		
A $\leftarrow$ STACK, SP $\leftarrow$ SP + 1				
N Z I C D V E				
+ + . . . .				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	PLA	68	1	4

## PLP

This instruction replaces the contents of the Processor Flags with the top value from the stack, and increments the value of the Stack Pointer by 1.

NOTE: This instruction does NOT replace the Extended Stack Disable Flag (E Flag), or the Software Interrupt Flag (B Flag)

<b>PLP : Pull Accumulator Register from the Stack</b>		<b>6502</b>		
A $\leftarrow$ STACK, SP $\leftarrow$ SP + 1				
N Z I C D V E				
+ + + + + + .				
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	PLP	28	1	4

# RLA

This instruction shifts the contents of the provided memory location one bit left. Bit 0 will be set to the current value of the Carry Flag, and the bit 7 will be shifted out into the Carry Flag. The result is then ANDed with the Accumulator.

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 7 of the value was set, prior to being shifted.

RLA : Rotate Left Memory, and AND with Accumulator					6502		
$M \leftarrow M \ll 1, C \leftarrow M(7), M(0) \leftarrow C, A \leftarrow A \text{ AND } M \ll 1$							
	N	Z	I	C	D	V	E
	+	+	.	+	.	.	.
Addressing Mode	Assembly	Code	Bytes	Bytes	Cycles		
(indirect,X)	RLA (\$nn,X)	23	2	2	8		
zero-page	RLA \$nn	27	2	2	5		
absolute	RLA \$nnnn	2F	3	3	6		
(indirect),Y	RLA (\$nn),Y	33	2	2	8		
zero-page,X	RLA \$nn,X	37	2	2	6		
absolute,Y	RLA \$nnnn,Y	3B	3	3	7		
absolute,X	RLA \$nnnn,X	3F	3	3	7		

# ROL

This instruction shifts either the Accumulator or contents of the provided memory location one bit left. Bit 0 will be set to the current value of the Carry Flag, and the bit 7 will be shifted out into the Carry Flag.

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 7 of the value was set, prior to being shifted.

**ROL : Rotate Left Memory or Accumulator****6502** $M \leftarrow M << 1, C \leftarrow M(7), M(0) \leftarrow C$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
zero-page	ROL \$nn	26	2	5
accumulator	ROL A	2A	1	2
absolute	ROL \$nnnn	2E	3	6
zero-page,X	ROL \$nn,X	36	2	6
absolute,X	ROL \$nnnn,X	3E	3	7

# ROR

This instruction shifts either the Accumulator or contents of the provided memory location one bit right. Bit 7 will be set to the current value of the Carry Flag, and the bit 0 will be shifted out into the Carry Flag

**Side effects**

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 7 of the value was set, prior to being shifted.

**ROR : Rotate Right Memory or Accumulator****6502** $M \leftarrow M >> 1, C \leftarrow M(0), M(7) \leftarrow C$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
zero-page	ROR \$nn	66	2	5
accumulator	ROR A	6A	1	2
absolute	ROR \$nnnn	6E	3	6
zero-page,X	ROR \$nn,X	76	2	6
absolute,X	ROR \$nnnn,X	7E	3	7

# RRA

This instruction shifts either the contents of the provided memory location one bit right. Bit 7 will be set to the current value of the Carry Flag, and the bit 0 will be shifted out into the Carry Flag. The result is added to the Accumulator.

## Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if the addition results in an overflow in the Accumulator.

<b>RRA : Rotate Right Memory, and Add to Accumulator</b>					<b>6502</b>
$M \leftarrow M \gg 1, C \leftarrow M(0), M(7) \leftarrow C, A \leftarrow A + M \gg 1 + C$					
<b>N Z I C D V E</b>					
		+	+	.	.
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	RRA (\$nn,X)	63	2	8	
zero-page	RRA \$nn	67	2	5	
absolute	RRA \$nnnn	6F	3	6	
(indirect),Y	RRA (\$nn),Y	73	2	8	
zero-page,X	RRA \$nn,X	77	2	6	
absolute,Y	RRA \$nnnn,Y	7B	3	7	
absolute,X	RRA \$nnnn,X	7F	3	7	

# RTI

This instruction pops the processor flags from the stack, and then pops the Programme Counter (PC) register from the stack, allowing an interrupted programme to resume.

- The 6502 Processor Flags are restored from the stack.
- Neither the B (Software Interrupt) nor E (Extended Stack) flags are set by this instruction.

<b>RTI : Return From Interrupt</b>	<b>6502</b>			
$P \leftarrow \text{STACK}, PC \leftarrow \text{STACK}, SP \leftarrow SP + 3$				
<b>N Z I C D V E</b>				
	+ . + + + .			
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	RTI	40	1	6

## RTS

This instruction adds optional argument to the Stack Pointer (SP) Register, and then pops the Programme Counter (PC) register from the stack, allowing a routine to return to its caller.

<b>RTS : Return From Subroutine</b>	<b>6502</b>			
$PC \leftarrow \text{STACK} + N, SP \leftarrow SP + 2 + N$				
<b>N Z I C D V E</b>				
	. . . . .			
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	RTS	60	1	6

## SAX

This instruction acts as a combination of AND and CMP. The result is stored in the X Register. Because it includes functionality from CMP rather than SBC, the Carry Flag is not used in the subtraction, although it is modified by the instruction.

NOTE: This instruction is affected by the status of the Decimal Flag.

### Side effects

- The N flag will be set if the result of  $A - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of  $A - M$  is zero or positive, i.e., if A is not less than M, else it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, else it will be cleared. If the flag is set, this indicates that a signed overflow has occurred.
- The Z flag will be set if the result of  $A - M$  is zero, else it will be cleared.

**SAX : AND Accumulator and X, and Subtract Without Carry 6502** $X \leftarrow (A \text{ AND } X) - \text{Value}$ 

N	Z	I	C	D	V	E
+	+	.	+	.	+	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
(indirect,X)	SAX (\$nn,X)	83	2	6
zero-page	SAX \$nn	87	2	3
absolute	SAX \$nnnn	8F	3	4
zero-page,Y	SAX \$nn,Y	97	2	4

**SBC**

This instruction performs  $A - M - 1 + C$ , and sets the processor flags accordingly. The result is stored in the Accumulator Register.

NOTE: This instruction is affected by the status of the Decimal Flag.

**Side effects**

- The N flag will be set if the result of  $A - M$  is negative, i.e. bit 7 is set in the result, else it will be cleared.
- The C flag will be set if the result of  $A - M$  is zero or positive, i.e., if A is not less than M, else it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, else it will be cleared. If the flag is set, this indicates that a signed overflow has occurred.
- The Z flag will be set if the result of  $A - M$  is zero, else it will be cleared.

**SBC : Subtract With Carry****6502** $A \leftarrow -M - 1 + C$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	+	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>	
(indirect,X)	SBC (\$nn,X)	E1	2	6	
zero-page	SBC \$nn	E5	2	3	
immediate	SBC #\$nn	E9	2	2	
immediate	SBC #\$nnn	EB	2	2	
absolute	SBC \$nnnn	ED	3	4	
(indirect),Y	SBC (\$nn),Y	F1	2	5	<i>p</i>
zero-page,X	SBC \$nn,X	F5	2	4	
absolute,Y	SBC \$nnnn,Y	F9	3	4	<i>p</i>
absolute,X	SBC \$nnnn,X	FD	3	4	<i>p</i>

p Add one cycle if indexing crosses a page boundary.

**SBX**

This instruction loads the X Register with the binary AND of the Accumulator Register and X Register, less the immediate argument.

NOTE: The subtraction effect in this instruction is due to CMP , not . Thus the Negative Flag is set according to the function of CMP, not SBC. That is, the carry flag is not used in the calculation.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if the result is zero or positive, else it will be cleared.

**SBX : AND and Subtract****6502** $X \leftarrow (A \text{ AND } X) - V$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	+	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
immediate	SBX #\$nn	CB	2	2

# SEC

This instruction sets the Carry Flag.

## Side effects

- The C flag is set.

SEC : Set Carry Flag		6502						
C ← 1		N	Z	I	C	D	V	E
Addressing Mode	Assembly Code	Bytes		Bytes		Bytes		Bytes
implied	SEC	38	1	1	2	1	2	1

# SED

This instruction sets the Decimal Flag. Binary arithmetic will now use Binary-Coded Decimal (BCD) mode.

NOTE: The C64's interrupt handler does not clear the Decimal Flag, which makes it dangerous to set the Decimal Flag without first setting the Interrupt Disable Flag.

## Side effects

- The D flag is set.

SED : Set Decimal Flag		6502						
D ← 1		N	Z	I	C	D	V	E
Addressing Mode	Assembly Code	Bytes		Bytes		Bytes		Bytes
implied	SED	F8	1	1	2	1	2	1

# SEI

This instruction sets the Interrupt Disable Flag. Normal (IRQ) interrupts will no longer be able to occur. Non-Maskable Interrupts (NMI) will continue to occur, as their name suggests.

## Side effects

- The I flag is set.

<b>SEI : Set Interrupt Disable Flag</b>		6502						
$I \leftarrow 1$		N	Z	I	C	D	V	E
Addressing Mode	Assembly	Code	Bytes	Cycles				
implied	SEI	78	1	2				

## SHA

NOTE: This instruction is unstable on many 6502 processors, and should be avoided.

This instruction stores the binary AND of the contents of the Accumulator Register, X Register and the third byte of the instruction into the indicated location.

<b>SHA : Store binary AND of A, X and 3rd Instruction Byte</b>		6502						
$M \leftarrow A \text{ AND } X \text{ AND } B3$		N	Z	I	C	D	V	E
Addressing Mode	Assembly	Code	Bytes	Cycles				
(indirect),Y	SHA (\$nn),Y	93	2	6				
absolute,Y	SHA \$nnnn,Y	9F	3	5				

## SHX

NOTE: This instruction is unstable on many 6502 processors, and should be avoided.

This instruction stores the binary AND of the contents of the X Register and the third byte of the instruction into the indicated location.

<b>SHX : Store Binary AND of X Register and 3rd Instruction Byte</b>	<b>6502</b>
$M \leftarrow X \text{ AND } B3$	

N	Z	I	C	D	V	E
.	.	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
absolute,Y	SHX \$nnnn,Y	9E	3	5

## SHY

NOTE: This instruction is unstable on many 6502 processors, and should be avoided.

This instruction stores the binary AND of the contents of the Y Register and the third byte of the instruction into the indicated location.

<b>SHY : Store Binary AND of Y Register and 3rd Instruction Byte</b>	<b>6502</b>
$M \leftarrow Y \text{ AND } B3$	

N	Z	I	C	D	V	E
.	.	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
absolute,X	SHY \$nnnn,X	9C	3	5

## SLO

This instruction shifts either contents of the provided memory location one bit left, and then ORs the result with the Accumulator Register, and places the result in the Accumulator.

### Side effects

- The N flag will be set if the result is negative, i.e., if bit 7 of the Accumulator is set after the instruction completes, else it will be cleared.
- The Z flag will be set if the Accumulator contains \$00 after the instruction has completed, else it will be cleared.
- The C flag will be set if bit 7 of the memory contents was set, prior to being shifted.

**SLO : Shift Left and OR****6502** $M \leftarrow M \ll 1, A \leftarrow M \ll 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
(indirect,X)	SLO (\$nn,X)	03	2	8
zero-page	SLO \$nn	07	2	5
absolute	SLO \$nnnn	0F	3	6
(indirect),Y	SLO (\$nn),Y	13	2	8
zero-page,X	SLO \$nn,X	17	2	6
absolute,Y	SLO \$nnnn,Y	1B	3	7
absolute,X	SLO \$nnnn,X	1F	3	7

**SRE**

This instruction shifts the contents of the provided memory location one bit right. Bit 7 will be set to zero, and the bit 0 will be shifted out into the Carry Flag. The result is exclusive ORed with the Accumulator and stored in the Accumulator.

**Side effects**

- The N flag will be set if the result is negative, i.e., if bit 7 is set after the operation, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.
- The C flag will be set if bit 0 of the value was set, prior to being shifted.

**SRE : Logical Shift Right and Exclusive OR with Accumulator    6502** $M \leftarrow M \gg 1, A \leftarrow A \text{ } XOR \text{ } M \gg 1$ 

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	+	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
(indirect,X)	SRE (\$nn,X)	43	2	8
zero-page	SRE \$nn	47	2	5
absolute	SRE \$nnnn	4F	3	6
(indirect),Y	SRE (\$nn),Y	53	2	8
zero-page,X	SRE \$nn,X	57	2	6
absolute,Y	SRE \$nnnn,Y	5B	3	7
absolute,X	SRE \$nnnn,X	5F	3	7

# STA

This instruction stores the contents of the Accumulator Register into the indicated location.

STA : Store Accumulator		6502			
M ← A		N	Z	I	C D V E
		.	.	.	.
Addressing Mode	Assembly	Code	Bytes	Cycles	
(indirect,X)	STA (\$nn,X)	81	2	6	
zero-page	STA \$nn	85	2	3	
absolute	STA \$nnnn	8D	3	4	
(indirect),Y	STA (\$nn),Y	91	2	6	
zero-page,X	STA \$nn,X	95	2	4	
absolute,Y	STA \$nnnn,Y	99	3	5	
absolute,X	STA \$nnnn,X	9D	3	5	

# STX

This instruction stores the contents of the X Register into the indicated location.

STX : Store X Register		6502			
M ← X		N	Z	I	C D V E
		.	.	.	.
Addressing Mode	Assembly	Code	Bytes	Cycles	
zero-page	STX \$nn	86	2	3	
absolute	STX \$nnnn	8E	3	4	
zero-page,Y	STX \$nn,Y	96	2	4	

# STY

This instruction stores the contents of the Y Register into the indicated location.

<b>STY : Store Y Register</b>	<b>6502</b>			
M ← Y				
	<b>N Z I C D V E</b>			
	.			
	.			
	.			
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
zero-page	STY \$nn	84	2	3
absolute	STY \$nnnn	8C	3	4
zero-page,X	STY \$nn,X	94	2	4

## TAS

NOTE: This monstrosity of an instruction, aside from being devoid of any conceivable useful purpose is unstable on many 6502 processors and should therefore also be avoided for that reason, if you had not already been put off.

### Side effects

- Remarkably, despite the over complicated operation that it performs, it modifies none of the processor flags.
- Loss of sanity if you attempt to use it, or even figure out exactly how it works.

<b>TAS : Munge X Register and Stack Pointer</b>	<b>6502</b>			
SP ← A AND X, M ← (A AND X) AND B3				
	<b>N Z I C D V E</b>			
	.			
	.			
	.			
<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
absolute,Y	TAS \$nnnn,Y	9B	3	5

## TAX

This instruction loads the X Register with the contents of the Accumulator Register.

### Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TAX : Transfer Accumulator Register into the X Register 6502**

X ← A

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	TAX	AA	1	2

## TAY

This instruction loads the Y Register with the contents of the Accumulator Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TAY : Transfer Accumulator Register into the Y Register 6502**

Y ← A

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	TAY	A8	1	2

## TSX

This instruction loads the X Register with the contents of the Stack Pointer High (SPL) Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TSX : Transfer Stack Pointer High Register into the X Register 6502**

X ← SPH

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	TSX	BA	1	2

**TXA**

This instruction loads the Accumulator Register with the contents of the X Register.

**Side effects**

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

**TXA : Transfer X Register into the Accumulator Register 6502**

A ← X

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
+	+	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	TXA	8A	1	2

**TXS**

This instruction sets the low byte of the Stack Pointer (SPL) register to the contents of the X Register.

**TXS : Transfer X Register into Stack Pointer Low Register 6502**

SPL ← X

<b>N</b>	<b>Z</b>	<b>I</b>	<b>C</b>	<b>D</b>	<b>V</b>	<b>E</b>
.	.	.	.	.	.	.

<b>Addressing Mode</b>	<b>Assembly</b>	<b>Code</b>	<b>Bytes</b>	<b>Cycles</b>
implied	TXS	9A	1	2

# TYA

This instruction loads the Accumulator Register with the contents of the Y Register.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

TYA : Transfer Y Register into the Accumulator Register 6502					
A ← Y					
N Z I C D V E					
		+	+	.	.
Addressing Mode	Assembly	Code	Bytes	Cycles	
implied	TYA	98	1	2	

# XAA

This instruction loads the Accumulator Register with the binary AND of the X Register and the immediate mode argument.

NOTE: This instruction is unstable on many 6502 processors, and should not be used.

## Side effects

- The N flag will be set if the result is negative, else it will be cleared.
- The Z flag will be set if the result is zero, else it will be cleared.

XAA : Transfer X into the Accumulator and AND with operand 6502					
A ← X AND VALUE					
N Z I C D V E					
		+	+	.	.
Addressing Mode	Assembly	Code	Bytes	Cycles	
immediate	XAA #\$nn	8B	2	2	



## APPENDIX

# F018-Compatible Direct Memory Access (DMA) Controller

- **Audio DMA**
- **MEGA65 Enhanced DMA Jobs**
- **F018 “DMAgic” DMA Controller**
- **MEGA65 DMA Controller Extensions**
- **Unimplemented Functionality**



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 DMAagic 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 implement a 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.
- **Per-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.
- **Background Audio DMA** The MEGA65 includes background audio DMA capabilities similar to the Amigaseries of computers. Key differences are that the MEGA65 can use either 8 or 16 bit samples, supports very high sample rates up to approximately 1 MHz, has 256 volume settings per channel, and no inter-channel modulation.

# AUDIO DMA

The MEGA65 includes four channels of DMA-driven audio playback that can be used in place of the direct digital audio registers at \$D6F8-\$D6FB. That is, you must select which of these two sources to feed to the audio cross-bar mixer. This is selected via the AUDEN signal (\$D711 bit 7), which simultaneously enables the audio DMA function in the processor, as well as instructing the audio cross-bar mixer to use the audio from this instead of the \$D6F8-\$D6FB digital audio registers. If you wish to have no other audio than the audio DMA channels, the audio cross-bar mixer can be bypassed, and the DMA audio played at full volume by setting the NOMIX signal (\$D711 bit 4). In that mode no audio from the SIDs, FM, microphones or other sources will be available. All other bits in \$D711 should ordinarily be left clear, i.e., write \$80 to \$D711 to enable audio DMA.

Two channels form the left digital audio channel, and the other two channels form the right digital audio channel. It is these left and right channels that are then fed into the MEGA65's audio cross-bar mixer.

As the DMA controller is part of the processor of the MEGA65, and the MEGA65 does not have reserved bus slots for multi-media operations, the MEGA65 uses idle CPU cycles to perform background DMA. This requires that the MEGA65 CPU be set to the "full speed" mode, i.e., approximately 40MHz. In this mode, there is a wait-state whenever reading an operand from memory. Thus each instruction that loads a byte from memory will create one implicit audio DMA slot. This is rarely a problem in practice, except if the processor idles in a very tight loop. To ensure that audio continues to play in the background, such loops should include a read instruction, such as:

```
loop: LDA $1234 // Ensure loop has at least one idle cycle for
               // audio DMA
    JMP loop
```

Each of the four DMA channels is configured using a block of 16 registers at \$D720, \$D730, \$D740 and \$D750, respectively. We will explain the registers for the first channel, channel 0, at \$D720 - \$D72F.

## Sample Address Management

To play an audio sample you must first supply the start address of the sample. This is a 24-bit address, and must be in the main chip memory of the MEGA65.

This is done by writing the address into \$D72A - \$D72C. This is the address of the first sample value that will be played. You must then provide the end address of the sample in \$D727 - \$D728. But note that this is only 16 bits. This is because the MEGA65 compares only the bottom 16 bits of the address when checking if it has reached the end of a sample. In practice, this means that samples cannot be more than 64KB in size. If the sample contains a section that should be repeated, then the start address of the repeating part should be loaded into \$D721 - \$D723, and the CH0LOOP bit should be set (\$D720 bit 6).

You can determine the current sample address at any time by reading the registers at \$D72A - \$D72C. But beware: These registers are not latched, so it is possible that the values may be updated as you read the registers, unless you stop the channel first by clearing the CH0EN signal.

## Sample Playback frequency and Volume

The MEGA65 controls the playback rate of audio DMA samples by using a 24-bit counter. Whenever the 24-bit counter overflows, the next sample is requested. Sample speed control is achieved by setting the value added to this counter each CPU cycle. Thus a value of \$FFFFFF would result in a sample rate of almost 40.5 MHz. In practice, sample rates above a few megahertz are not possible, because there are insufficient idle CPU cycles, and distorted audio will result. Even below this, care must be taken to ensure that idle cycles come sufficiently often and dispersed throughout the processor's instruction stream to prevent distortion. At typical sample rates below 16KHz and using 8 bit samples these effects are typically negligible for normal instruction streams, and so no special action is normally required for typical audio playback.

At the other end of the scale, sample rates as low as  $40.5\text{MHz}/2^{24} = 2.4$  samples per second are possible. This is sufficiently low enough for even the most demanding infra-sound applications.

Volume is controlled by setting \$D629. Maximum volume is obtained with the value \$FF, while a value of \$00 will effectively mute the channel. The first two audio channels are normally allocated to the left, and the second two to the right. However, the MEGA65 includes separate volume controls for the opposite channels. For example, to play audio DMA channel 0 at full volume on both left and right sides of the audio output, set both \$D729 and \$D71C to \$FF. This allows panning of the four audio DMA channels.

Both the frequency and volume can be freely adjusted while a sample is playing to produce various effects.

## Pure Sine Wave

Where it is necessary to produce a stable sine wave, especially at higher frequencies, there is a special mode to support this. By setting the CH0SINE signal, the audio channel will play a 32-byte 16-bit sine wave pattern. The sample addresses still need to be set, as though the sine wave table were located in the bottom 64 bytes of memory, as the normal address generation logic is used in this mode. However, no audio DMA fetches are performed when a channel is in this mode, thus avoiding all sources of distortion due to irregular spacing of idle cycles in the processor's instruction stream.

This can be used to produce sine waves in both the audible range, as well as well into the ultrasonic range, at frequencies exceeding 60,000Hz, provided that the MEGA65 is connected to an appropriately speaker arrangement.

## Sample playback control

To begin a channel playing a sample, set the CH0EN signal (\$D720 bit 7). The sample will play until its completion, unless the CH0LOOP signal has also been set. When a sample completes playing, the CH0STP flag will be set. The audio DMA subsystem cannot presently generate interrupts.

Unlike on the Amiga, the MEGA65 audio DMA system supports both 8 and 16-bit samples. It also supports packed 4-bit samples, playing either the lower or upper nybl of each sample byte. This allows two separate samples to occupy the same byte, thus effectively halving the amount of space required to store two equal length samples.

# MEGA65 ENHANCED DMA JOBS

The MEGA65's implementation of the DMAgic supports significantly enhanced DMA jobs. An enhanced DMA job is indicated by writing the low-byte of the DMA list address to \$D705 instead of to \$D700. The MEGA65 will then look for one or more *job option tokens* at the start of the DMA list. Those tokens will be interpreted, before executing the DMA job which immediately follows the *end of job options token* (\$00). Job option tokens which take an argument have the most-significant bit set, and always take a 1 byte option.

Job option tokens that take no argument have the most-significant-bit clear. Unsupported job option tokens are simply ignored. This allows for future revisions of the DMAgic to add support for additional options, without breaking backward compatibility.

The list of valid job option tokens is:

\$00	End of job option list
\$06	Enable use of transparent value
\$07	Disable use of transparent value
\$0A	Use 11-byte F011A DMA list format
\$0B	Use 12-byte F011B DMA list format
\$80	Source address bits 20 - 27
\$81	Destination address bits 20 - 27
\$82	Source skip rate (256 <sup>ths</sup> of bytes)
\$83	Source skip rate (whole bytes)
\$84	Destination skip rate (256 <sup>ths</sup> of bytes)
\$85	Destination skip rate (whole bytes)
\$86	Transparent value (bytes with matching value are not written)

## 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).
D702	55042	ADDRBANK	DMA list address bank (address bits 16 - 22). Writing clears \$D704.

## MEGA65 DMA CONTROLLER EXTENSIONS

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
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)

## UNIMPLEMENTED FUNCTIONALITY

The MEGA65's DMAgic does not currently support either memory-swap or mini-term operations.

Miniterms were intended for bitplane blitting, which is not required for the MEGA65 which offers greatly advanced character modes and stepped and fractional DMA address incrementing which allows efficient texture copying and scaling. Also there exists no known software which ever used this facility, and it remains uncertain if it was ever implemented in any revision of the DMAgic chip used in C65 prototypes.

The memory-swap operation is intended to be implemented, but can be worked around in the meantime by copying the first region to a 3rd region that acts as a temporary buffer, then copying the 2nd region to the 1st, and the 3rd to the 2nd.

# J 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  $\sim 8$ MHz with the VIC-II/III), which enables a wide range of new features.
- New 16-colour ( $16 \times 8$  pixels per character cell) and 256-colour ( $8 \times 8$  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 un-drawn and redrawn to animate them.
- **Multiple 8-bit colour play-fields** 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:

<b>First Value Hex (Decimal)</b>	<b>Second Value Hex (Decimal)</b>	<b>Effect</b>	<b>VIC-IV Specific?</b>
\$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 W
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
```

Line 10 of this programme checks whether the screen is a multiple of 2KiB. As the screen on the C64 is located at 1KiB, this test will fail, and execution will continue to line 20. Line 20 writes 1 to one of the VIC-II sprite position registers, 53248, before writing the MEGA65 knock to the key register, 53295. Line 30 writes to 53248 + 256, which on the C64 is a mirror of 53248, but on a MEGA65 with VIC-IV IO enabled will be one of the red palette registers. After writing to 53248 + 256, the programme checks if the register at 53248 has been modified by the write to 53248 + 256. If it has, then the two addresses point to the same register. This will happen on either a C64 or C65, but not on a computer with a VIC-IV. Thus if 53248 has not changed, we report that we have detected a VIC-IV. If writing to 53248 + 256 did change the value in register 53248, then we proceed to line 40, which writes to 53248 again, and this time writes the VIC-III knock to the key register. Line 50 is like line 30, but as it appears after a VIC-III knock, it allows the detection of a VIC-III. Finally, if neither a VIC-IV nor VIC-III is detected, we conclude that only a VIC-II must be present.

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**

## **Integrated Marvellous Digital Hookup™(IMDH™) Digital Video Output**

The MEGA65 features VGA analog video output and Integrated Marvellous Digital Hookup™ (IMDH™). This is different to existing common digital video standards in several key points:

1. We didn't invent a new connector for it: We instead used the most common digital video connector already in use. So your existing cables should work fine!
2. We didn't make it purposely incompatible with any existing digital video standard. So your existing TVs and monitors should work fine!
3. We don't engage in highway-robbery for other vendors to use the IMDH™ digital video standard, by trying to charge them \$10,000 every year, just for the permission to be able to sell a single device. This means that the MEGA65 is cheaper for you!
4. The IMDH™ standard does not allow content-protection or other sovereignty eroding flim-flam. If you produced the video, you can do whatever you like with it!

### **Connecting to Naughty Proprietary Digital Video Standards**

There are digital video standards that are completely backwards compared with IMDH™. Fortunately because of IMDH™'s open approach to interoperability, these should, in most cases, function with the MEGA65 without difficulty. Simply find a video cable fits the IMDH™ connector on the back of your MEGA65, and connect it to your MEGA65 and a TV, Monitor or Projector that has the same connector.

However, regrettably, not all manufacturers have submitted their devices for IMDH™ compliance testing with the MEGA65 team. This means that some TVs and Monitors are, unfortunately, not IMDH™ compliant. Thus while most TVs and Monitors will work with the MEGA65, you might find that you need to try a couple to get a satisfactory result. If you do find a monitor that

doesn't work with the MEGA65, please let us know, and also report the problem to the Monitor vendor, recommending that they submit their devices for IMDH™ compliance testing.

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 the non-proprietary IMDH™ interface. 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  $720 \times 576$  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 81MHz 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.

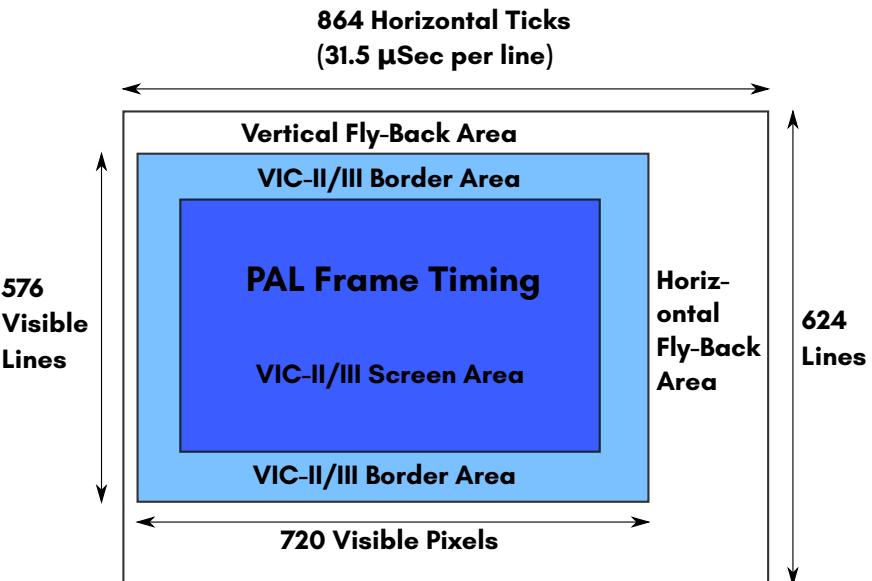
4. The  $1280 \times 200$  and  $1280 \times 400$  display modes of the VIC-III are not currently supported, as they cannot be meaningfully displayed on any modern monitor, and no software is known to support or use this feature.

## 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.

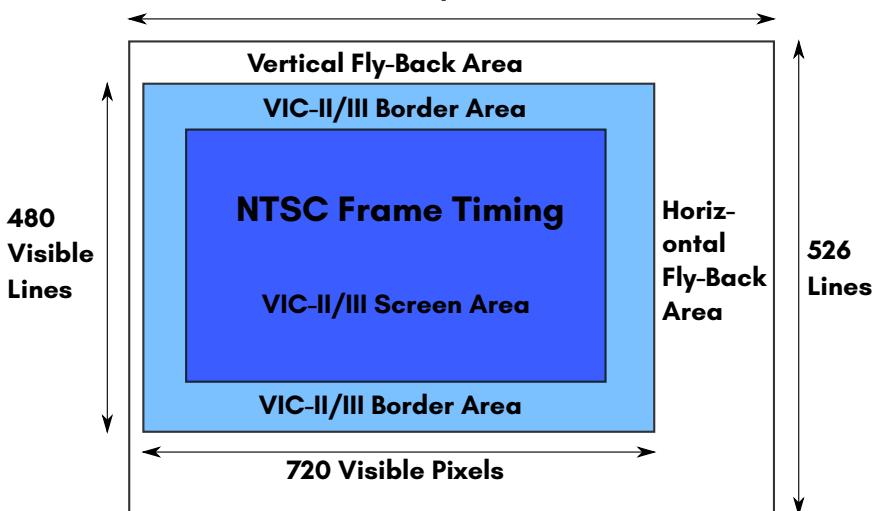
The PAL frame is constructed from 624 physical raster lines, consisting of 864 pixel clock ticks. The pixel clock is 27MHz, which is 1/3 the VIC-IV pixel clock. The visible frame is  $720 \times 576$  pixels, the entirety of which can be used in VIC-IV mode. In VIC-II and VIC-III modes, the border area reduces the usable size to  $640 \times 400$  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.



The NTSC frame is constructed from 526 physical raster lines, consisting of 858 pixel clock ticks. The pixel clock is 27MHz, which is 1/3 the VIC-IV pixel clock. The visible frame is  $720 \times 480$  pixels, the entirety of which can be used in VIC-IV mode. In VIC-II and VIC-III modes, the border area reduces the usable size to  $640 \times 400$  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.

**858 Horizontal Ticks**

(32  $\mu$ Sec per line)



As these HDTV video modes are not supported by all VGA monitors, a compatibility mode is included that provides a  $640 \times 480$  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 IMDH™-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("S"):REM ENABLE C65+MEGA65 IO  
20 NTSC=PEEK(53359)AND128  
30 IFNTSC>0THENPRINT"MEGA65 IS IN NTSC MODE"  
40 IFNTSC=0THENPRINT"MEGA65 IS IN PAL MODE"  
50 INPUT"SWITCH MODES (Y/N)? ",A$  
60 IFA$="Y"THENPOKE53359,PEEK(53359)+128-NTSC  
70 NTSC=PEEK(53359)AND128  
80 IFNTSC>0THENPRINT"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:

<b>Standard</b>	<b>Cycles per Raster</b>	<b>Physical Rasters per Frame</b>	<b>Logical Rasters per Frame</b>
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  $\sim 1\text{MHz}$  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  $49\,152 \div 64 = 768$ . 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 (53270 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, 53270, 53272 and 53297 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 chip-set to include so-called “chunky” modes, rather than just bitplane based 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 DMAagic to assist, scrolling a  $320 \times 200$  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  $8 \times 8$  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  $8 \times 8$  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:

<b>Bit(s)</b>	<b>Function</b>
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
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  $8 \times 8$  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  $80 \times 50$  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  $40 \times 25$  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  $40 \times 25$  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 play-fields on other computers, such as the Amiga, but without the disadvantages of bitplanes.

XXX Example program

## 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  $64 \text{ bits} \div 4 \text{ bits per pixel} = 16$  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:  $spritenumber \times 16 + nibblevalue$ , 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 MEGAB5 VIC-IV FEATUR
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 READNS:IFNS="END"THEN RETURN
920 GOSUB 1000
930 GOTO 910

1000 REM DECODE STRING OF NIBBLES IN NS AT ADDRESS AD
1010 L=LEN(NS)
1020 FOR I=1 TO (L/2+1):POKE AD+I,0
1030 FOR I= 1 TO L:N=ASC(MID$(NS,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 EFGHIJKLMNO"
2010 DATA "A A EEE EEE EEE EEE EEE EEE"
2020 DATA "A C A EEE EEE EEE EEE EEE EEE"
2030 DATA "A C C A EEE EEE EEE EEE EEE EEE"
2040 DATA "A C C C A EEE EEE EEE EEE EEE EEE"
2050 DATA "A C C C C A EEE EEE EEE EEE EEE EEE"
2060 DATA "A C C C C C A EEE EEE EEE EEE EEE EEE"
2070 DATA "A C C C C C C A EEE EEE EEE EEE EEE EEE"
2080 DATA "A C C C C C C C A EEE EEE EEE EEE EEE EEE"
2090 DATA "A C C C C C C C C A EEE EEE EEE EEE EEE EEE"
2100 DATA "A C C C C C C C C C A EEE EEE EEE EEE EEE EEE"
2110 DATA "A C C C C C C C C C C A EEE EEE EEE EEE EEE EEE"
2120 DATA "A C C C C C C C C C C C A EEE EEE EEE EEE EEE EEE"
2130 DATA "A C C C C C C C C C C C C A EEE EEE EEE EEE EEE EEE"
2140 DATA "A C C C C C C C C C C C C C A EEE EEE EEE EEE EEE EEE"
2150 DATA "A C C C C C C C C C C C C C C A EEE EEE EEE EEE EEE EEE"
```

# VIC-II / C64 REGISTERS

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
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					L.PY			
D015	53269					SE			
D016	53270	-	RST	MCM	CSEL			XSCL	
D017	53271					SEXY			
D018	53272		VS			CB			-
D019	53273		-			ILP	ISSC	ISBC	RIRQ
D01A	53274		-			MISSC	MISBC		MRIIRQ
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			

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
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
- **ILP** light pen indicate or acknowledge
- **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
- **SPRMCI** Sprite multi-colour 1
- **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				

continued ...

...continued

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
D025	53285						SPRMC0		
D026	53286						SPRMC1		
D02F	53295						KEY		
D030	53296	ROME	CROM9	ROMC	ROMA	ROM8	PAL	EXTSYNC	CRAM2K
D031	53297	H640	FAST	ATTR	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			
D200 - D2FF	53760 - 54015					PALGREEN			
D300 - D3FF	54016 - 54271					PALBLUE			

- **ATTR** Enable extended attributes and 8 bit colour entries
- **BOADEVN** - Bitplane 0 address, even lines

- **BOADODD** - Bitplane 0 address, odd lines
- **BOPIX** 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 Translater (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)
- **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)
- **V400** Enable 400 vertical pixels
- **VPOS** Bitplane Y Offset

# VIC-IV / MEGA65 SPECIFIC REGISTERS

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
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	SMTH	FCLRHI	FCLRLO	CHR16
D055	53333				SPRHGTEN				
D056	53334				SPRHGHT				
D057	53335				SPRX64EN				
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				

continued ...

...continued

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
D062	53346				SCRNPTR				
D063	53347				SCRNPTR				
D064	53348				COLPTR				
D065	53349				COLPTR				
D068	53352				CHARPTR				
D069	53353				CHARPTR				
D06A	53354				CHARPTR				
D06B	53355				SPR16EN				
D06C	53356				SPRPTRADR				
D06D	53357				SPRPTRADR				
D06E	53358	SPRPTR16			SPRPTRBNK				
D06F	53359	PALNTSC	VGAHDTV		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			Number					
D07C	53372	RESERVED		VSYNCP	HSYNCP		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
- **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)
- **Number** of text rows to display
- **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)
- **VGAHDTV** Select more VGA-compatible mode if set, instead of HD-MI/HDTV VIC-II cycle-exact frame timing. May help to produce a functional display on older VGA monitors.
- **VSYNCP** vsync polarity
- **VSYNDEL** VIC-IV VSYNC delay
- **XPOS** Read horizontal raster scan position

# K

## APPENDIX

# 6526 Complex Interface Adaptor (CIA) Registers

- CIA 6526 Registers
- CIA 6526 Hypervisor Registers



# CIA 6526 REGISTERS

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
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
- **FLG** 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

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
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

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
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<sub>A</sub>** Port A DDR
- **DDR<sub>B</sub>** 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

<b>HEX</b>	<b>DEC</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
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

# L

## APPENDIX L

# 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
- **RXOVERRUN** 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	HDSDA	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	JOYSWAP	OSKDEBUG				-
D615	54805	OSKEN				VIRTKEY1			
D616	54806	OSKALT				VIRTKEY2			
D617	54807	OSKTOP				VIRTKEY3			
D618	54808				KSCRNRATE				
D619	54809				UNUSED				
D61A	54810				UNUSED				
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)
- **HDSDL** HDMI I2C control interface SCL clock
- **HSDA** HDMI I2C control interface SDA data line
- **JOYSWAP** Exchange joystick ports 1 2
- **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)
- **OSKDEBUG** Debug OSK overlay (WRITE ONLY)
- **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.
- **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
- **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.
- **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)
- **UNUSED** port o output value
- **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 3rd virtual key press.

# M

## 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 G.

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	RXBUI	RXBMR	-
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:

```
POKE55012,1
```

<b>HEX</b>	<b>DEC</b>	<b>Signal</b>	<b>Description</b>
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.



# N

## 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

<b>Decimal</b>	<b>Binary</b>	<b>Hexadecimal</b>
0	%0	\$0
2	%10	\$2
4	%100	\$4
6	%110	\$6
8	%1000	\$8
10	%1010	\$A
12	%1100	\$C
14	%1110	\$E
16	%10000	\$10
18	%10010	\$12
20	%10100	\$14
22	%10110	\$16
24	%11000	\$18
26	%11010	\$1A
28	%11100	\$1C
30	%11110	\$1E

<b>Decimal</b>	<b>Binary</b>	<b>Hexadecimal</b>
32	%100000	\$20
34	%100010	\$22
36	%100100	\$24
38	%100110	\$26
40	%101000	\$28
42	%101010	\$2A
44	%101100	\$2C
46	%101110	\$2E
48	%110000	\$30
50	%110010	\$32
52	%110100	\$34
54	%110110	\$36
56	%111000	\$38
58	%111010	\$3A
60	%111100	\$3C
62	%111110	\$3E

<b>Decimal</b>	<b>Binary</b>	<b>Hexadecimal</b>
64	%1000000	\$40
66	%1000010	\$42
68	%1000100	\$44
70	%1000110	\$46
72	%1001000	\$48
74	%1001010	\$4A
76	%1001100	\$4C
78	%1001110	\$4E
80	%1010000	\$50
82	%1010010	\$52
84	%1010100	\$54
86	%1010110	\$56
88	%1011000	\$58
90	%1011010	\$5A
92	%1011100	\$5C
94	%1011110	\$5E

<b>Decimal</b>	<b>Binary</b>	<b>Hexadecimal</b>
96	%1100000	\$60
98	%1100010	\$62
100	%1100100	\$64
102	%1100110	\$66
104	%1101000	\$68
106	%1101010	\$6A
108	%1101100	\$6C
110	%1101110	\$6E
112	%1110000	\$70
114	%1110010	\$72
116	%1110100	\$74
118	%1110110	\$76
120	%1111000	\$78
122	%1111010	\$7A
124	%1111100	\$7C
126	%1111110	\$7E

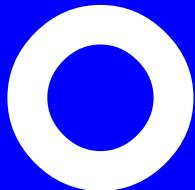
<b>Decimal</b>	<b>Binary</b>	<b>Hexadecimal</b>
128	%10000000	\$80
130	%10000010	\$82
132	%10000100	\$84
134	%10000110	\$86
136	%10001000	\$88
138	%10001010	\$8A
140	%10001100	\$8C
142	%10001110	\$8E
144	%10010000	\$90
146	%10010010	\$92
148	%10010100	\$94
150	%10010110	\$96
152	%10011000	\$98
154	%10011010	\$9A
156	%10011100	\$9C
158	%10011110	\$9E

<b>Decimal</b>	<b>Binary</b>	<b>Hexadecimal</b>
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

<b>Decimal</b>	<b>Binary</b>	<b>Hexadecimal</b>
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

<b>Decimal</b>	<b>Binary</b>	<b>Hexadecimal</b>
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 5 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.

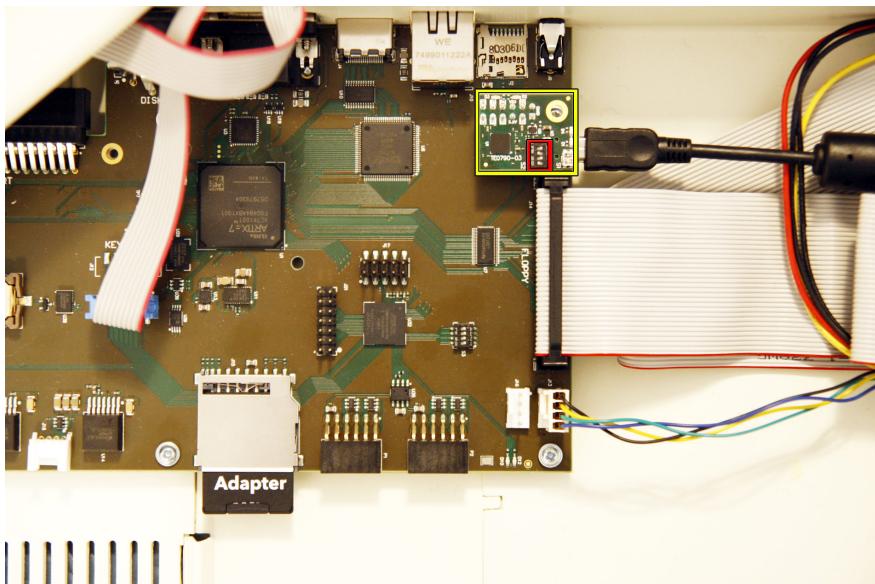
Be prepared to install many gigabytes of software on a Linux or Windows PC, before you will be able to write programs for the FPGAs and CPLDs in the MEGA65. Also, “compiling” complex designs can take up to several hours, depending on the speed and memory capacity of your computer! We recommend a computer 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 Internet.

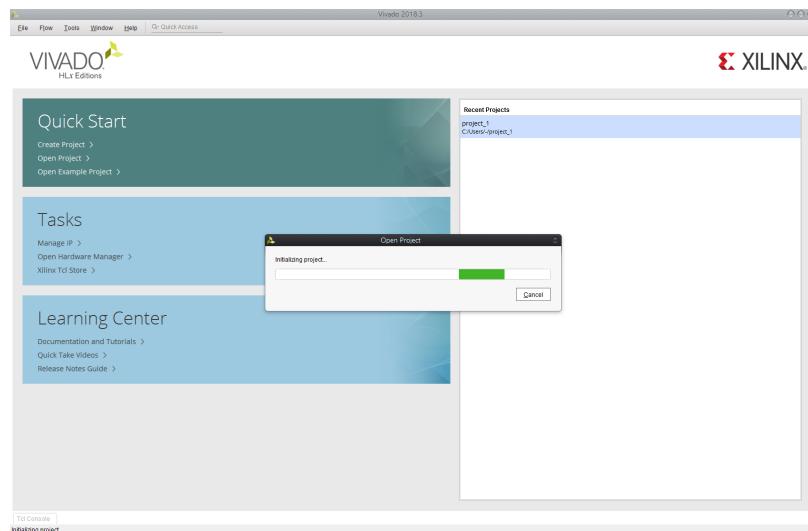


Figure O.1: Step 1: To access the Hardware Manager, open a project in VIVADO or create an empty one, if you do not have any projects yet.

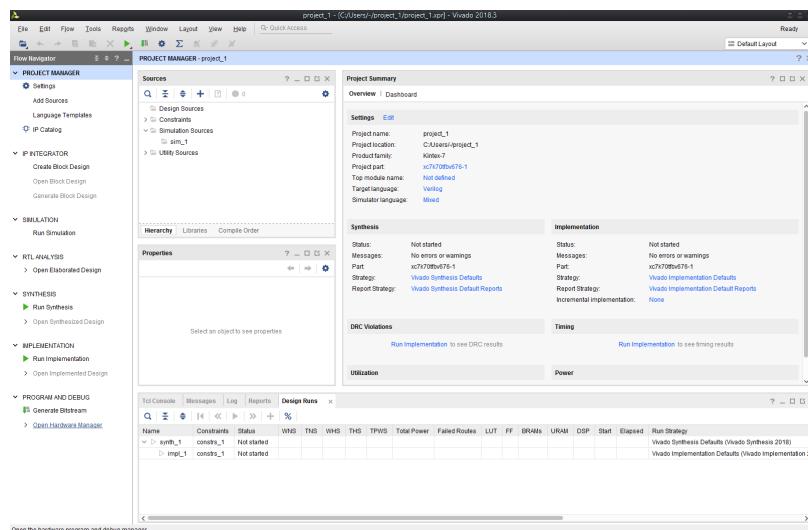


Figure O.2: Step 2: In the left column, select "Open Hardware Manager" at the very bottom.

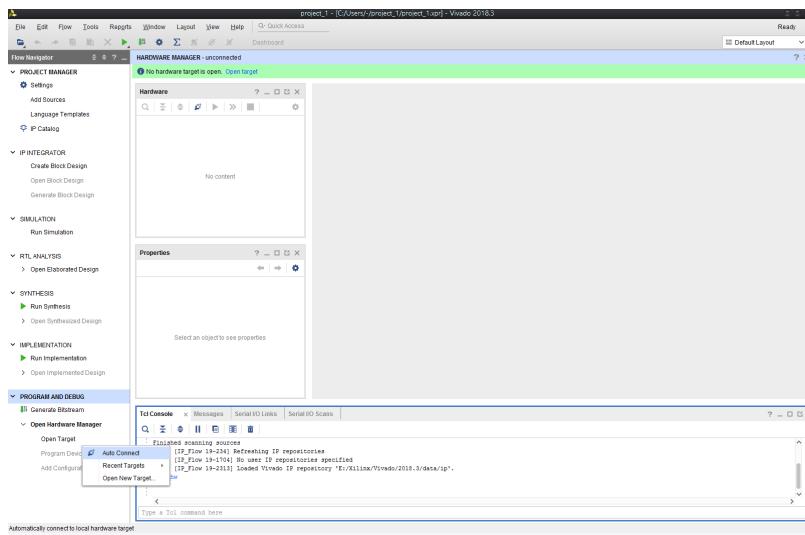


Figure O.3: Step 3: To connect to FPGA under "Hardware Manager", choose "Open Target", then "Auto Connect".

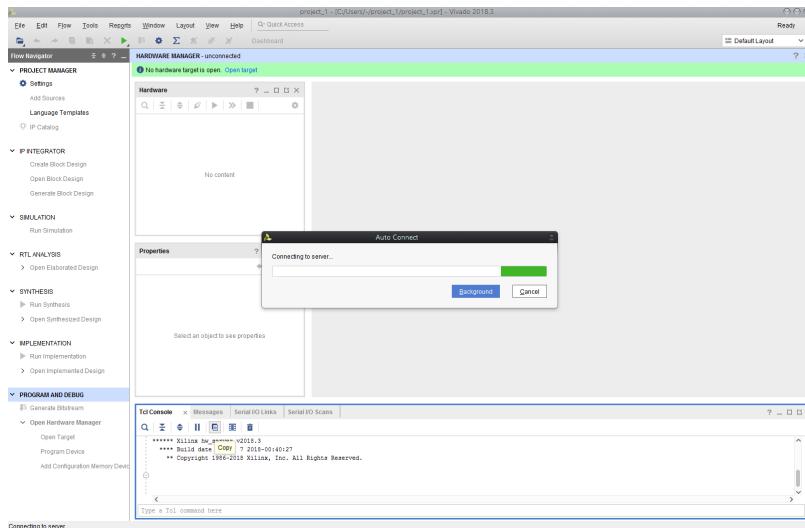


Figure O.4: Step 4: Wait a moment, "Connecting to server..." should automatically close without dropping an error to the console.

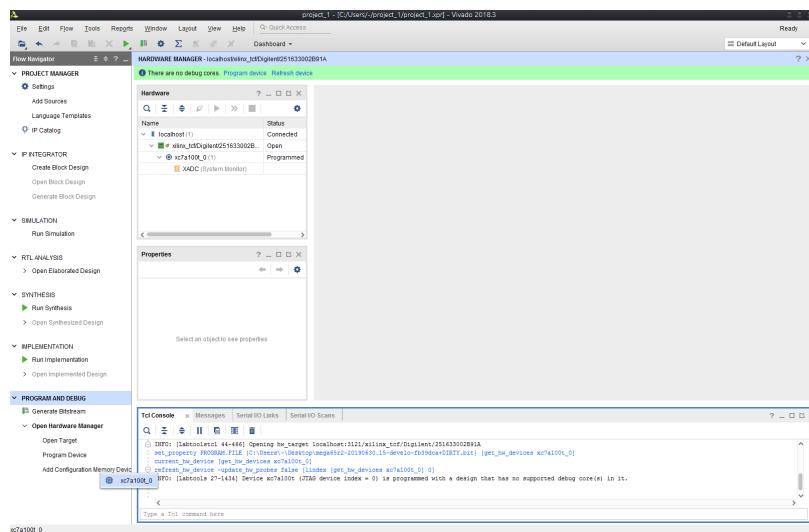


Figure O.5: Step 5: Under "Hardware Manager", choose "Add Configuration Memory Device", then "xc7a100t\_0".

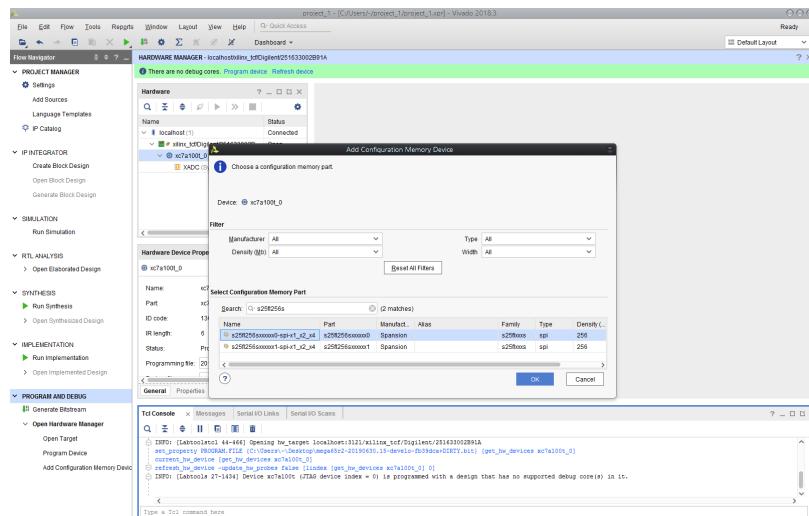


Figure O.6: Step 6: Select Memory Part: In the newly opened dialogue, type "S25fl256s" (without quotes), then select "s25fl256xxxxxxxx0-spi-x1\_x2\_x4" (the upper one) and click "OK".

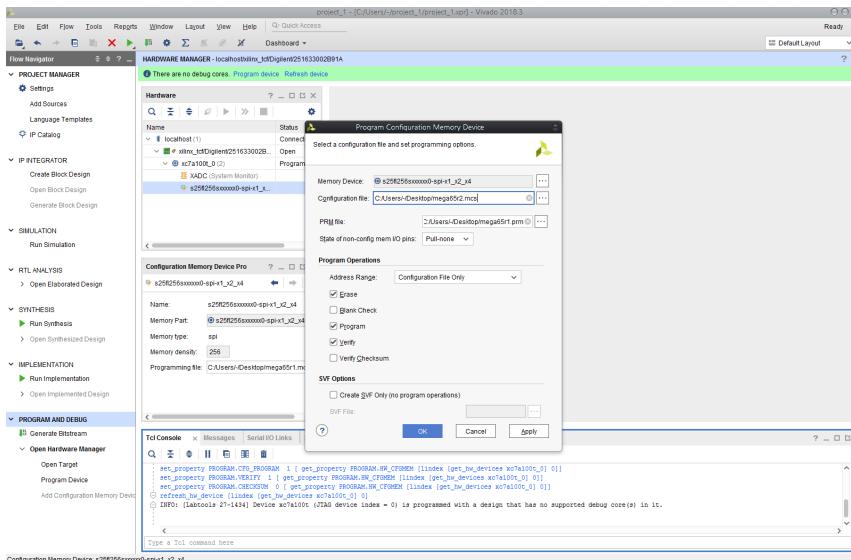


Figure O.7: Step 7: Set programming options: In the next dialogue, choose your local Configuration file, namely a bitstream with file suffix ".mcs". Leave all other parameters as they are (see O.7).

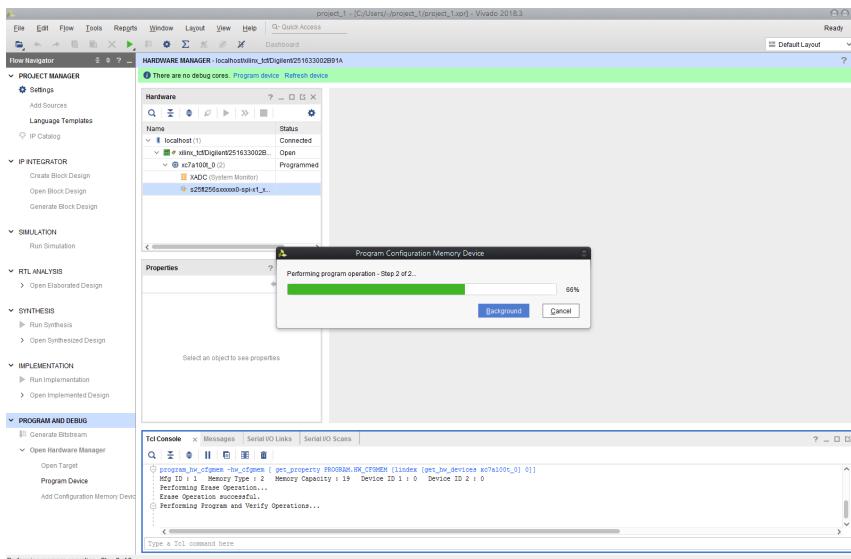


Figure O.8: Step 8: Patiently wait for the programming to finish. This can take several minutes as the Vivado software erases and then reprograms the flash memory that is used to initialise the FPGA on power-up.

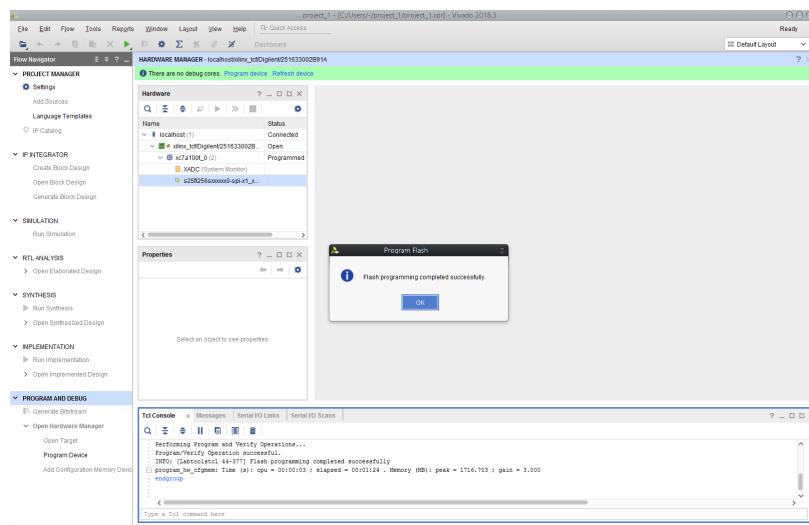


Figure O.9: Step 9: If your screen looks like O.9, your new bistream has been successfully flashed into the Artix 100T FPGA!

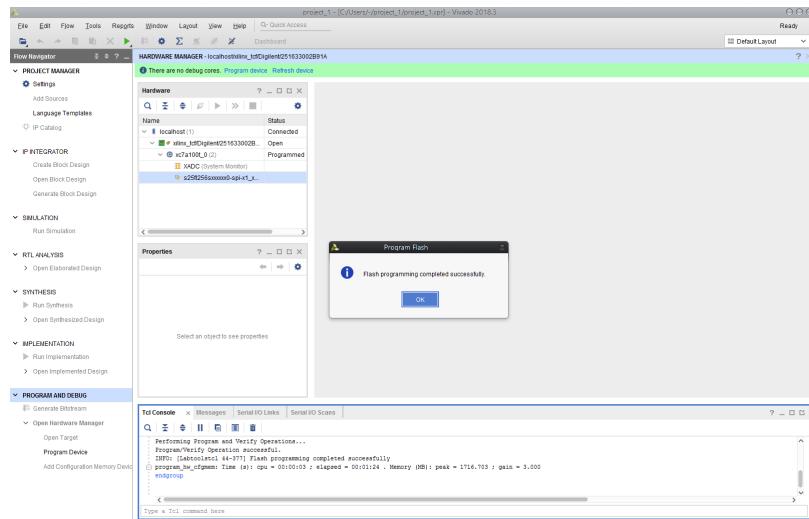
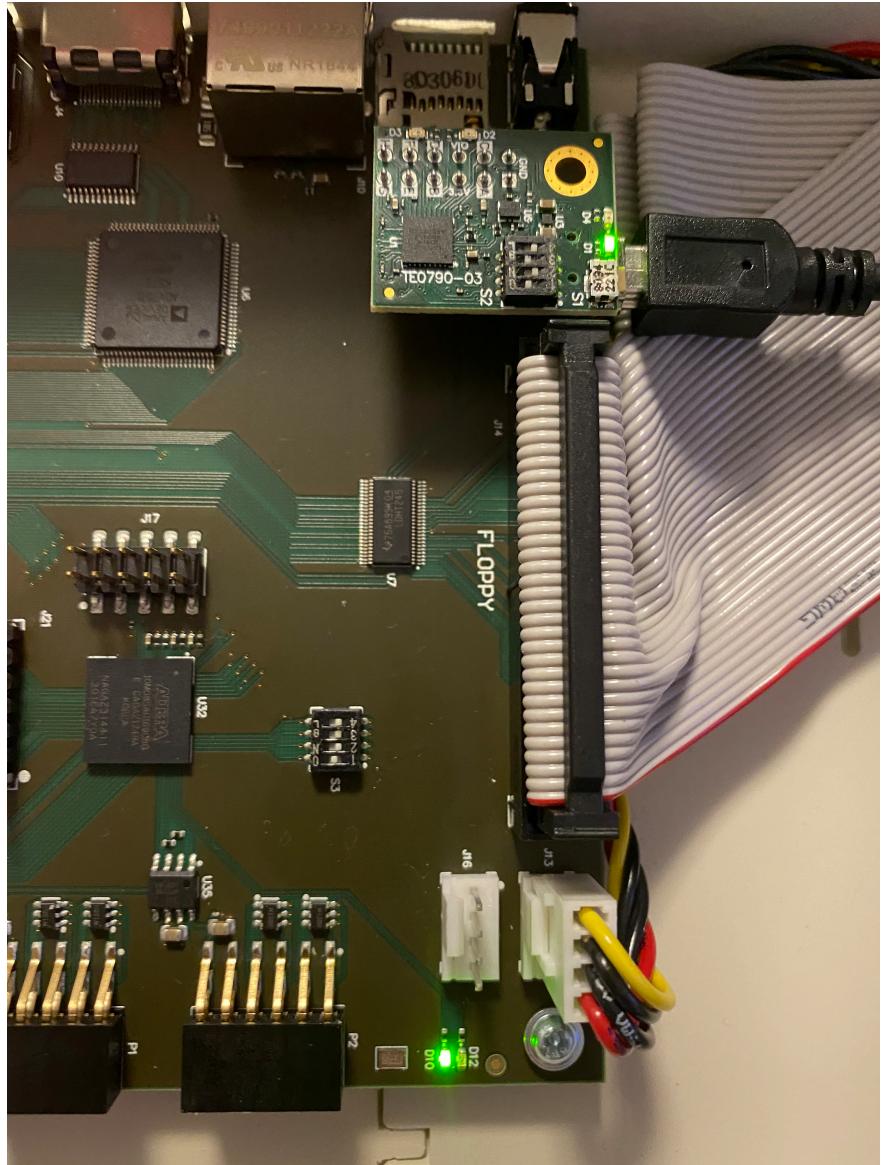


Figure O.10: Step 10: If you want to reflash the FPGA, you might find the "Add Configuration Memory Device" option in step 5 greyed out. Instead, select "s25fl256xxxxxxxx0-spi-x1\_x2-x4" in the "Hardware" window, press right mouse button and select "Program Configuration Memory Device" to flash.

# 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 Mainboard 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.

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 Internet.

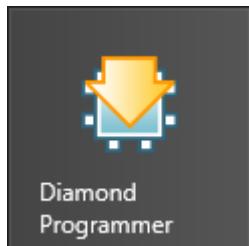


Figure O.11: Step 1: Open DIAMOND PROGRAMMER: Select "Create a new project from a JTAG scan". If entry under "Cable:" is empty, click "Detect Cable".

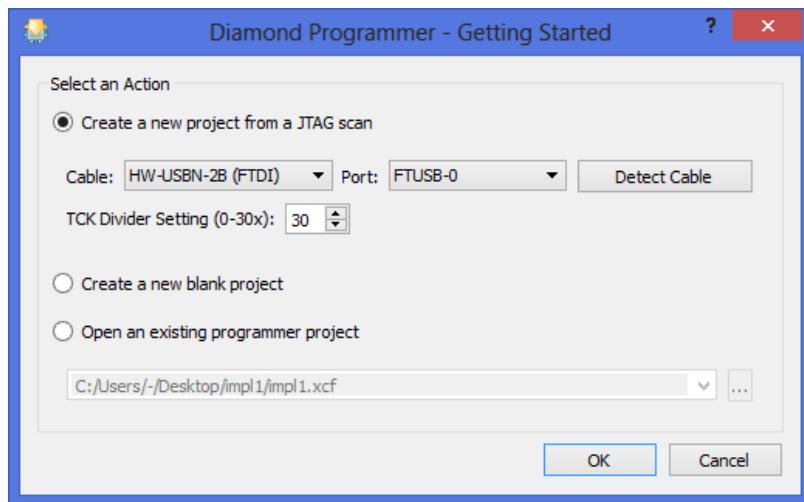


Figure O.12: Step 2: Create a new project: If dialog "Programmer: Multiple Cables Detected" appears, select the first entry ("Location 0000") and click "OK".

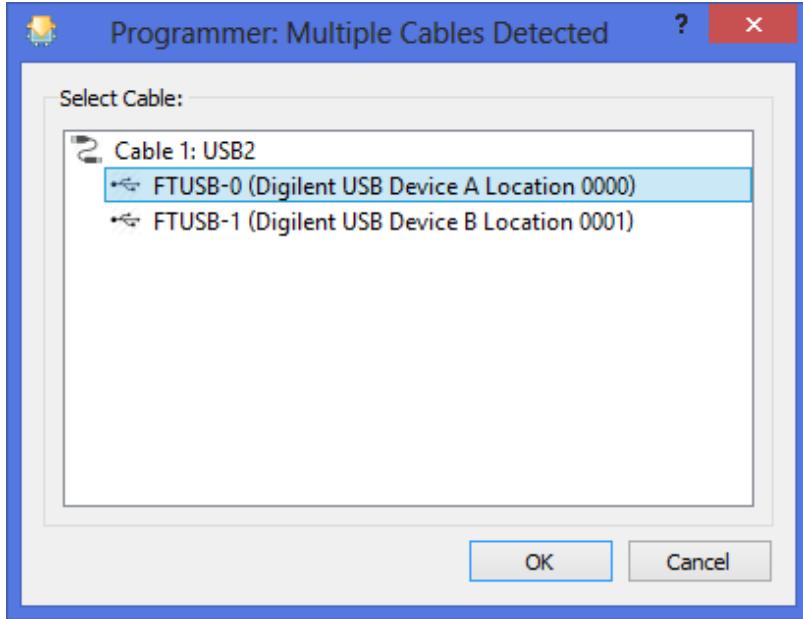


Figure O.13: Step 3: Select cable: You have now created a new project which should display "MachXO2" under "Device Family" and "LCMXO2-1200HC" under "Device"

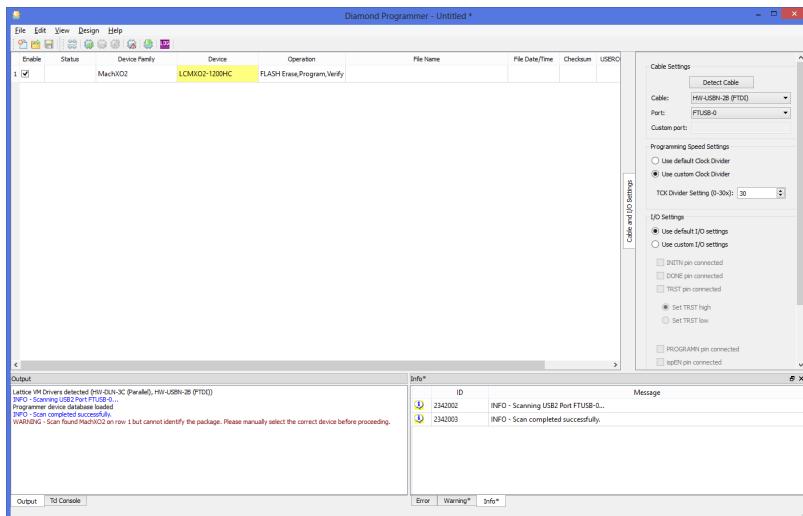


Figure O.14: Step 4: New Diamond Programmer project: Choose "File" then "Open File" to load the DIAMOND PROGRAMMER project with the MEGA65 keyboard firmware update.

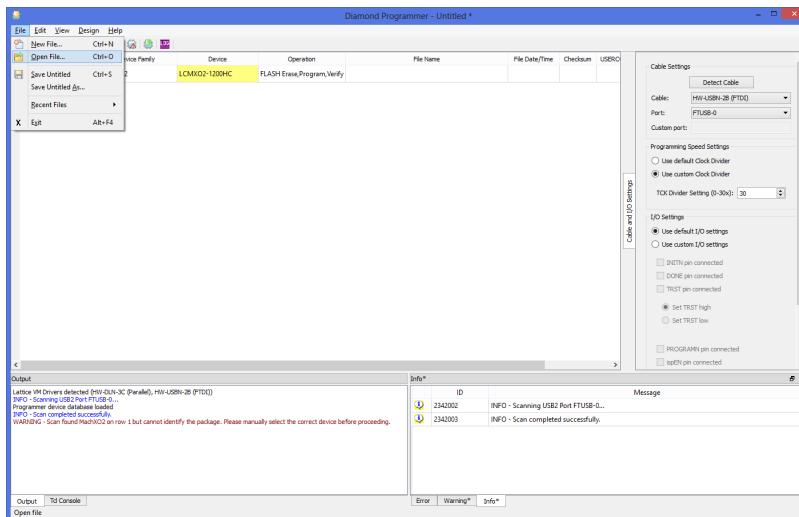


Figure O.15: Step 5: Open project: Navigate into the folder with the extracted MEGA65 keyboard firmware files you have received and select the file ending with ".xcf".

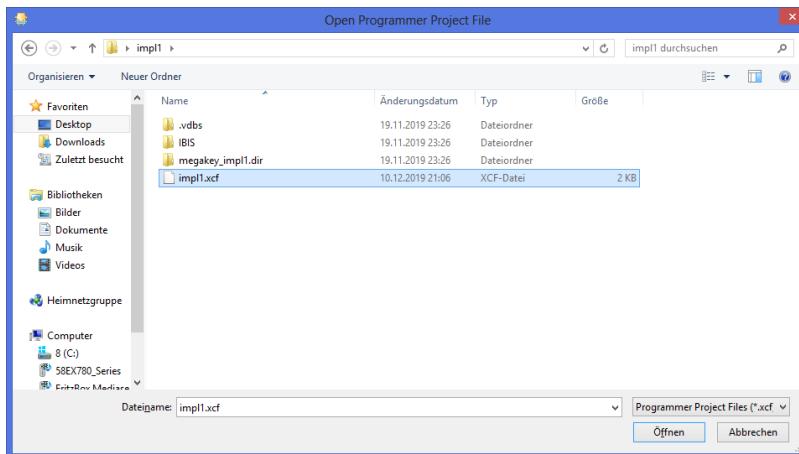


Figure O.16: Step 6: Select project file: Click the three dots under "File Name" to set the correct path and find the file ending with ".jed".

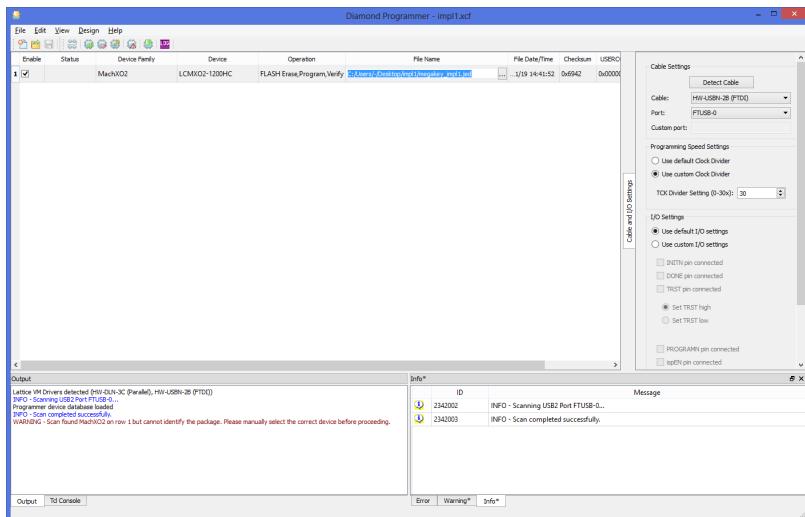


Figure O.17: Step 7: Choose correct path of .jed file: Select the file ending with ".jed" and click "OK".

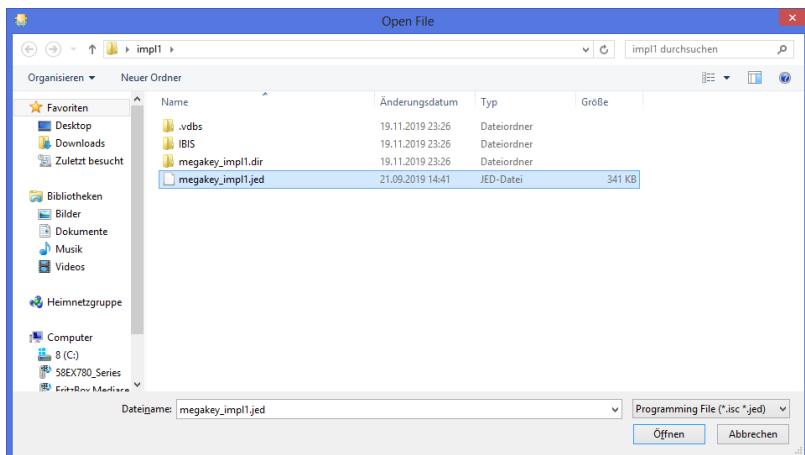


Figure O.18: Step 8: Select .jed file: Click on the icon with the green arrow facing down "PROGRAM", which looks similar to the DIAMOND PROGRAMMER program icon.

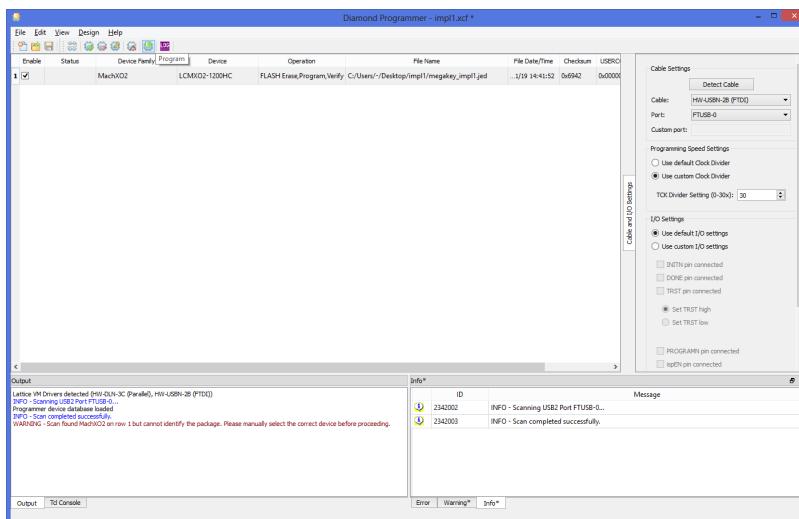


Figure O.19: Step 9: Select cable: After a moment the Output window should display "INFO - Operation: successful." and the "Status" cell should go green (does not always happen).

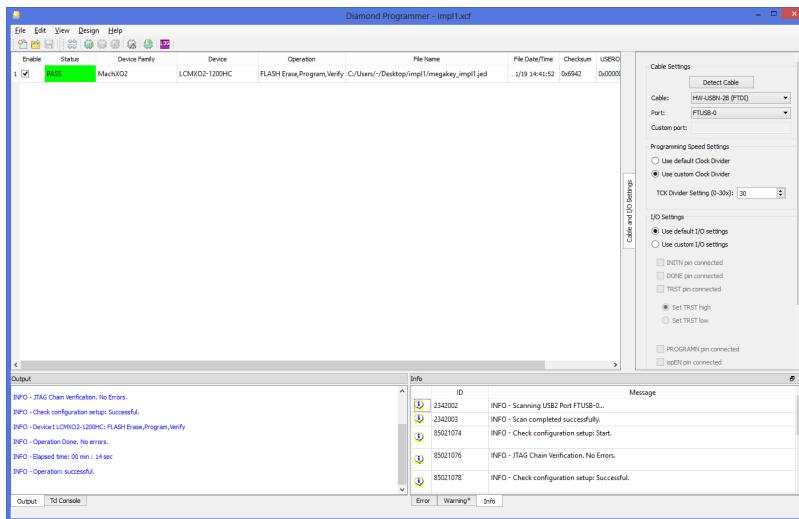
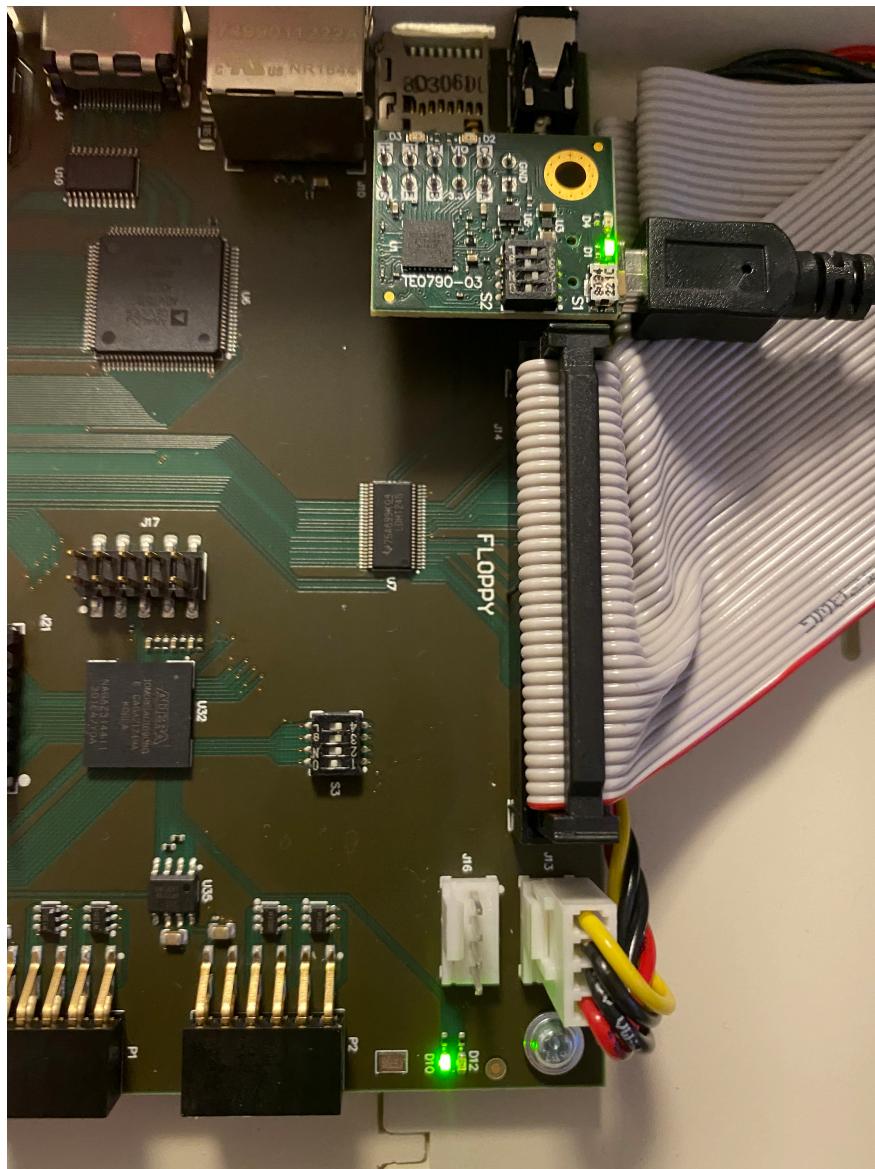


Figure O.20: Step 10: Operation successful: You have now successfully flashed the MEGA65 keyboard. If you wish you can save the project now for later use.

# 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 Internet.



Figure O.21: Step 1: Open Quartus Prime Programmer Lite Edition: Click the "Hardware Setup" button in the top left corner of the Quartus Prime Programmer window.

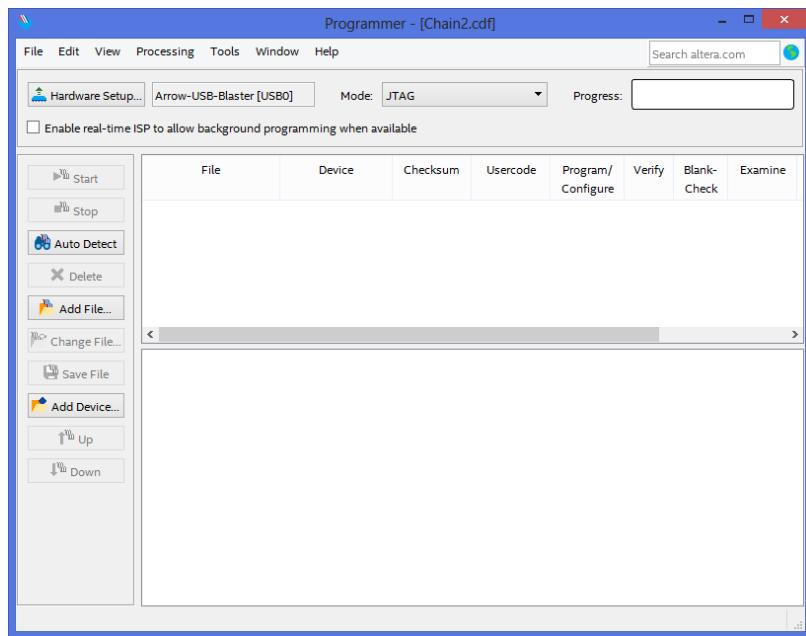


Figure O.22: Step 2: Enter Hardware Setup: In the newly appeared window under "Currently selected hardware" choose "Arrow-USB-Blaster". If "Arrow-USB-Blaster" does not appear, verify cable and drivers being correctly installed.

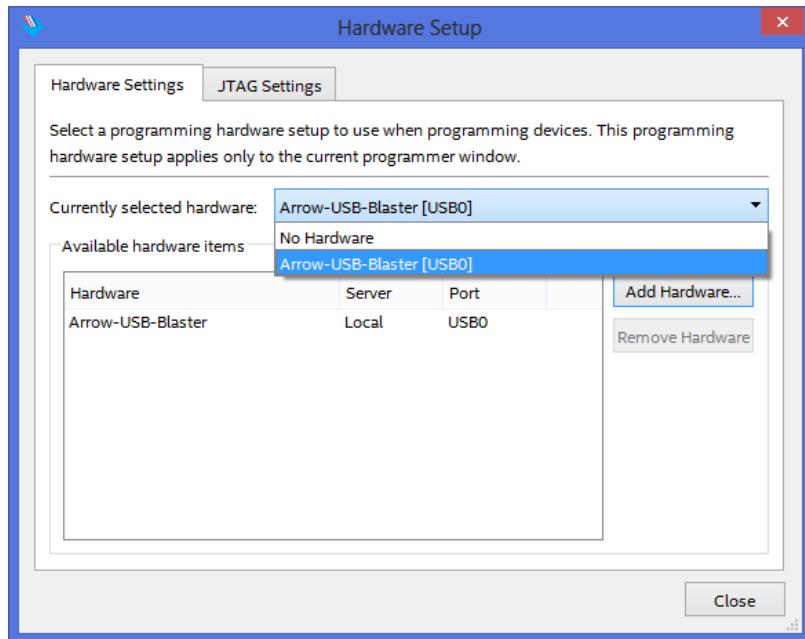


Figure O.23: Step 3: Select Arrow USB-Blaster: Click the "Add File" button from the left row and choose the latest ".pof" file. Then click "Open".

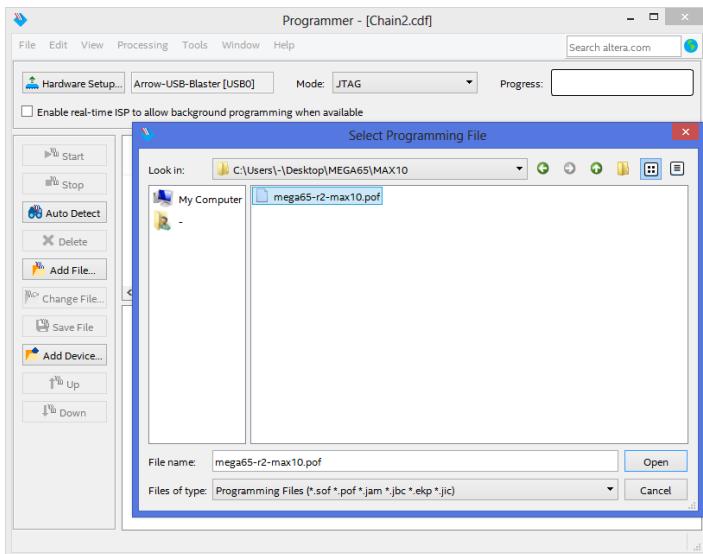


Figure O.24: Step 4: Select Programming File: Tick at least the three boxes under "Program/Configure". Also enabling all boxes under "Verify" and "Blank-Check" will make the process more reliable.

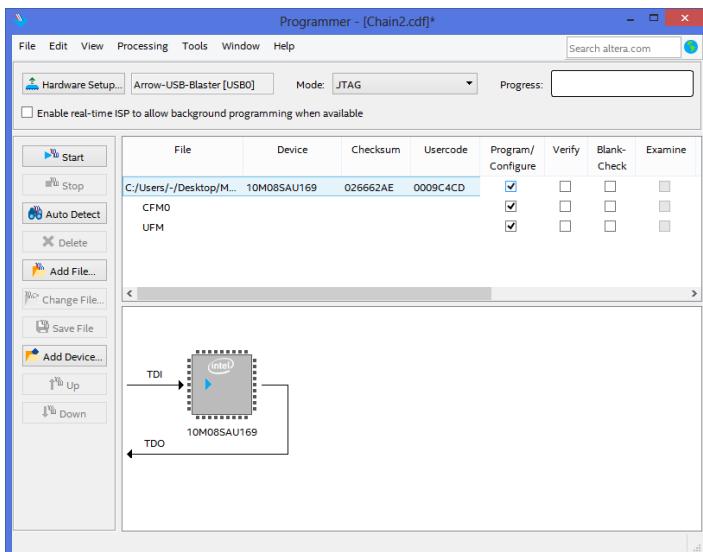


Figure O.25: Step 5: Select Program/Configure Options

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.

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.

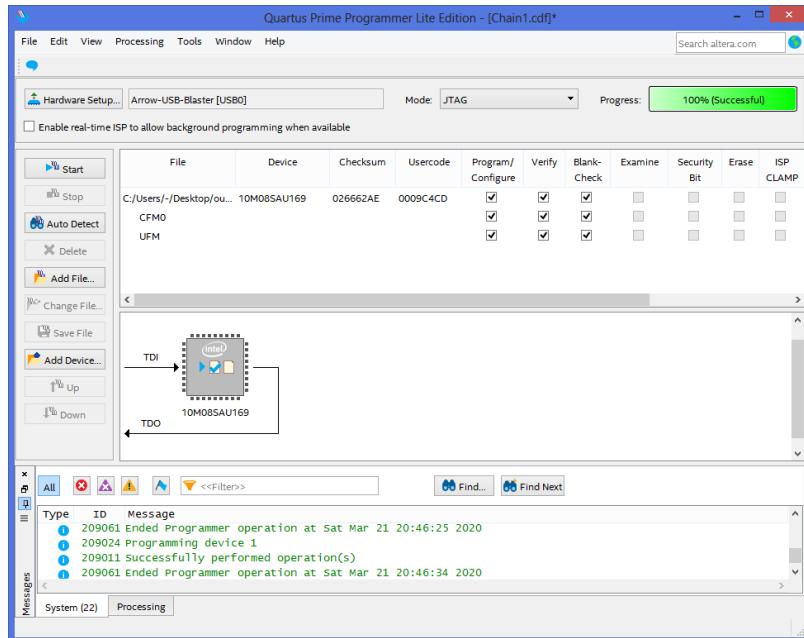


Figure O.26: Step 6: Programming successful

# P

## APPENDIX

### Model Specific Features

- MEGA65 Desktop Computer, Revision 2 onwards
- MEGAphone 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 un-hidden, 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 joy-pad 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.



# APPENDIX Q

## 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 apologies 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**
- **Hintsteiner, 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



, (comma), 76, 94  
: (colon), 76, 94  
<> (not equal to), 85

ABS, 156  
ADC, 448, 510  
ALR, 511  
ANC, 511  
AND, 99, 157, 448, 512  
APPEND, 158  
ARR, 513  
ASC, 159  
ASL, 449, 514  
ASR, 450  
ASW, 451  
ATN, 160  
AUTO, 161

BACKGROUND, 99, 162  
BACKUP, 163  
BANK, 164  
BASIC 10 Commands, 317  
    APPEND, 158  
    AUTO, 161  
    BACKGROUND, 99, 162  
    BACKUP, 163  
    BANK, 164, 390  
    BEGIN, 165  
    BEND, 166  
    BLOAD, 167  
    BOOT, 168  
    BORDER, 99, 169  
    BOX, 170  
    BSAVE, 171  
    BUMP, 172  
    BVERIFY, 173  
    CATALOG, 174  
    CHANGE, 175  
    CHAR, 176  
    CIRCLE, 178  
    CLOSE, 179  
    CLR, 180  
    CMD, 181

COLLECT, 182  
COLLISION, 183  
COLOR, 184  
CONCAT, 185  
CONT, 93, 186  
COPY, 187  
CURSOR, 189  
DATA, 190  
DCLEAR, 191  
DCLOSE, 192  
DEF FN, 194  
DELETE, 195  
DIM, 196  
DIR, 197  
DISK, 198  
DLOAD, 199  
DMA, 200, 390  
DMODE, 201  
DO, 202  
DOPEN, 203  
DPAT, 205  
DSAVE, 206  
DVERIFY, 207  
ELLIPSE, 209  
ELSE, 210  
END, 93, 211  
ENVELOPE, 212  
ERASE, 213  
EXIT, 216  
FAST, 218  
FILTER, 219  
FIND, 220  
FOR, 61, 222  
FOREGROUND, 99, 223  
GET, 225  
GET#, 226  
GETKEY, 227  
GO64, 228  
GOSUB, 229  
GOTO, 101, 230  
GRAPHIC, 231  
HEADER, 232

HELP, 233  
HIGHLIGHT, 235  
IF, 84, 236  
INPUT, 71, 78, 237  
INPUT#, 238  
INSTR, 239  
KEY, 242  
LET, 70, 245  
LINE, 246  
LIST, 65, 66, 247  
LOAD, 248  
LOCATE, 249  
LOOP, 251  
MONITOR, 255  
MOUSE, 256  
MOVSPR, 257  
NEW, 258  
NEXT, 259  
OFF, 261  
ON, 262  
OPEN, 264  
PAINT, 266  
PALETTE, 267  
PEN, 269  
PLAY, 270  
POLYGON, 274  
PRINT, 65, 277  
PRINT USING, 279  
PRINT#, 278  
PUDEF, 281  
READ, 284  
RECORD, 285  
REM, 286  
RENAME, 287  
RENUMBER, 92, 288  
RESTORE, 289  
RESUME, 290  
RETURN, 291  
RMOUSE, 294  
RREG, 296  
RUN, 300  
SAVE, 301  
SCNCLR, 302  
SCRATCH, 303  
SCREEN, 304  
SET, 305  
SLEEP, 308  
SLOW, 309  
SOUND, 310  
SPRCOLOR, 312  
SPRITE, 313  
SPRSAV, 314  
STEP, 67, 316  
STOP, 93  
SYS, 319  
TEMPO, 323  
THEN, 84, 324  
TO, 325  
TRAP, 326  
TROFF, 327  
TRON, 328  
TYPE, 329  
UNTIL, 330  
USING, 331  
VERIFY, 335  
VOL, 336  
WAIT, 337  
WHILE, 338  
WINDOW, 339

BASIC 10 Functions

ABS, 156  
ASC, 159  
ATN, 160  
CHR\$, 177  
COS, 188  
DEC, 193  
ERR\$, 215  
EXP, 217  
FN, 194, 221  
FRE, 224  
HEX\$, 234  
INT, 240  
JOY, 241  
LEFT\$, 243

LEN, 244	BBR7, 454
LOG, 250	BBS0, 455
LPEN, 252	BBS1, 455
MID\$, 253	BBS2, 456
MOD, 254	BBS3, 456
PEEK, 268	BBS4, 456
POINTER, 272	BBS5, 457
POKE, 273	BBS6, 457
POS, 275	BBS7, 458
POT, 276	BCC, 458, 514
RCLR, 282	BCS, 458, 515
RDOT, 283	BEGIN, 165
RGR, 292	BEND, 166
RIGHT\$, 293	BEQ, 459, 515
RND, 100, 295	BIT, 459, 515
RSPCOLOR, 297	BLOAD, 167
RSPPOS, 298	blocked, 75
RSPRITE, 299	BMI, 460, 516
SGN, 306	BNE, 460, 516
SIN, 307	BOOT, 168
SPC, 311	BORDER, 99, 169
SQR, 315	BOX, 170
STR\$, 318	BPL, 461, 517
TAB, 321	BRA, 461
TAN, 322	BRK, 462, 517
USR, 333	BSAVE, 171
VAL, 334	BSR, 462
BASIC 10 Operators	BUMP, 172
AND, 99, 157	BVC, 463, 518
NOT, 260	BVERIFY, 173
OR, 265	BVS, 463, 518
XOR, 340	CATALOG, 174
BASIC 10 System Variables	CHANGE, 175
EL, 208	CHAR, 176
ER, 214	character, 67
BBR0, 452	character set, 67
BBR1, 452	CHR\$, 177
BBR2, 452	CIRCLE, 178
BBR3, 453	CLC, 464, 518
BBR4, 453	CLD, 464, 519
BBR5, 454	CLE, 464
BBR6, 454	CLI, 465, 519

CLOSE, 179  
CLR, 180  
CLV, 465, 520  
CMD, 181  
CMP, 466, 520, 542  
COLLECT, 182  
COLLISION, 183  
COLOR, 184  
CONCAT, 185  
CONT, 93, 186  
context dependent, 73  
COPY, 187  
copyright, ii  
COS, 188  
CPX, 467, 521  
CPY, 467, 522  
CPZ, 468  
CURSOR, 189

DATA, 190  
DCLEAR, 191  
DCLOSE, 192  
DCP, 522  
DEC, 193, 468, 523  
DEF FN, 194  
DELETE, 195  
DEW, 469  
DEX, 470, 523  
DEY, 470, 524  
DEZ, 471  
digital video, 569  
DIM, 196  
DIR, 197  
Direct Mode, 72  
DISK, 198  
DLOAD, 199  
DMA, 200  
DMODE, 201  
DO, 202  
DOPEN, 203  
DPAT, 205  
DSAVE, 206  
DVERIFY, 207

EL, 208  
ELLIPSE, 209  
ELSE, 210  
END, 93, 211  
ENVELOPE, 212  
EOM, 471  
EOR, 472, 524  
ER, 214  
ERASE, 213  
ERR\$, 215  
Errors  
    Extra Ignored, 76  
    Illegal Direct, 72  
    Syntax, 58  
    Type mismatch, 71  
EXIT, 216  
EXP, 217  
Extra Ignored, 76

FAST, 218  
FILTER, 219  
FIND, 220  
FN, 194, 221  
FOR, 61, 222  
FOREGROUND, 99, 223  
FRE, 224

Games  
    Guess the number, 87  
GET, 225  
GET#, 226  
GETKEY, 227  
GO64, 228  
GOSUB, 229  
GOTO, 101, 230  
GRAPHIC, 231  
    Guess the number, 87

HEADER, 232  
HELP, 233  
HEX\$, 234  
HIGHLIGHT, 235

IF, 84, 236

Illegal Direct Error, 72  
IMDH™, 569  
INC, 473, 525  
INPUT, 71, 78, 237  
INPUT#, 238  
INSTR, 239  
INT, 240  
Integrated Marvellous Digital Hookup™, 569  
INW, 474  
INX, 475, 525  
INY, 475, 526  
INZ, 475  
IO  
    blocking, 75  
ISC, 526  
JMP, 476, 527  
JOY, 241  
JSR, 476, 527  
KEY, 242  
KIL, 528  
LAS, 529  
LAX, 530  
LDA, 477, 530  
LDX, 478, 531  
LDY, 478, 532  
LDZ, 479  
LEFT\$, 243  
LEN, 244  
LET, 70, 245  
LINE, 246  
Lines  
    editing, 79  
    renumbering, 92  
    replacing, 79  
LIST, 65, 66, 247  
LOAD, 248  
LOCATE, 249  
LOG, 250  
LOOP, 251  
LPEN, 252  
LSR, 479, 532  
MAP, 480  
MID\$, 253  
MOD, 254  
MONITOR, 255  
MOUSE, 256  
MOVSPR, 257  
name spaces, 71  
NEG, 481  
NEW, 258  
NEXT, 259  
NO SCROLL, 102  
NOP, 533  
NOT, 260  
not equal, 85  
OFF, 261  
ON, 262  
OPEN, 264  
operators  
    relational, 85  
OR, 265  
ORA, 481, 534  
PAINT, 266  
PALETTE, 267  
PEEK, 268  
PEN, 269  
PHA, 482, 535  
PHP, 482, 535  
PHW, 483  
PHX, 483  
PHY, 484  
PHZ, 484  
PLA, 484, 536  
PLAY, 270  
PLP, 485, 536  
PLX, 485  
PLY, 485  
PLZ, 486

POINTER, 272	RRA, 539
POKE, 273	RREG, 296
POLYGON, 274	RSPCOLOR, 297
POS, 275	RSPPOS, 298
POT, 276	RSprite, 299
PRINT, 65, 277	RTI, 491, 539
PRINT USING, 279	RTS, 491, 540
PRINT#, 278	RUN, 300
Programmes	
editing, 79	SAVE, 301
replacing lines, 79	SAX, 540
PUDEF, 281	SBC, 492, 541, 542
quote mode, 89	SBX, 542
RCLR, 282	scientific notation, 100
RDOT, 283	SCNCLR, 302
READ, 284	SCRATCH, 303
RECORD, 285	SCREEN, 304
relational operators, 85	SEC, 493, 543
REM, 286	SED, 493, 543
RENAME, 287	SEE, 494
RENUMBER, 92, 288	SEI, 494, 543
RESTORE, 289	SET, 305
RESUME, 290	SGN, 306
RETURN, 291	SHA, 544
RGR, 292	SHX, 544
RIGHT\$, 293	SHY, 545
RLA, 537	SIN, 307
RMB0, 486	SLEEP, 308
RMB1, 486	SLO, 545
RMB2, 487	SLOW, 309
RMB3, 487	SMB0, 495
RMB4, 487	SMB1, 495
RMB5, 488	SMB2, 496
RMB6, 488	SMB3, 496
RMB7, 488	SMB4, 496
RMOUSE, 294	SMB5, 497
RND, 295	SMB6, 497
RND(), 100	SMB7, 497
ROL, 489, 537	SOUND, 310
ROR, 490, 538	SPC, 311
ROW, 490	SPRCOLOR, 312
	SPRITE, 313
	SPRSAV, 314

SQR, 315  
SRE, 546  
STA, 498, 547  
STEP, 67, 316  
STOP, 93, 317  
STR\$, 318  
string, 67  
STX, 498, 547  
STY, 499, 547  
STZ, 499  
SYNTAX ERROR, 58  
SYS, 319  
  
TAB, 321, 499  
TAN, 322  
TAS, 548  
TAX, 500, 548  
TAY, 500, 549  
TAZ, 501  
TBA, 501  
TEMPO, 323  
THEN, 84, 324  
TO, 325  
TRAP, 326  
TRB, 502  
TROFF, 327  
TRON, 328  
TSB, 503  
TSX, 503, 549  
  
TSY, 504  
TXA, 504, 550  
TXS, 505, 550  
TYA, 505, 551  
TYPE, 329  
Type mismatch error, 71  
TYS, 506  
TZA, 506  
  
unequal, 85  
UNTIL, 330  
USING, 331  
USR, 333  
  
VAL, 334  
variable, 62  
    numeric, 69  
    string, 69  
VERIFY, 335  
VOL, 336  
  
WAIT, 337  
Warnings  
    Extra Ignored, 76  
WHILE, 338  
WINDOW, 339  
  
XAA, 551  
XOR, 340



# 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 multi-line keys: **RUN STOP** **CLR HOME** **NO SCRL** **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

\texttrademark

## Amiga™ computers

# KEYBOARD KEYS



MEGA key looks like this.

**NORMAL SHIFT**

**BIG SHIFT**

Text to the left and text to the right.

**SHIFT** **CTRL** **9**  **RETURN**

**\*** **←** **↑** **→** **↓**

# SCREEN OUTPUT

```
↑↑I10 INPUT A$  
↑↑I20 PRINT "YOU TYPED: ";A$  
↑↑I30 PRINT  
↑↑I40 GOTO 10  
↑↑IRUN  
↑↑I? MEGA 65  
↑↑IVOU TYPED: MEGA 65
```

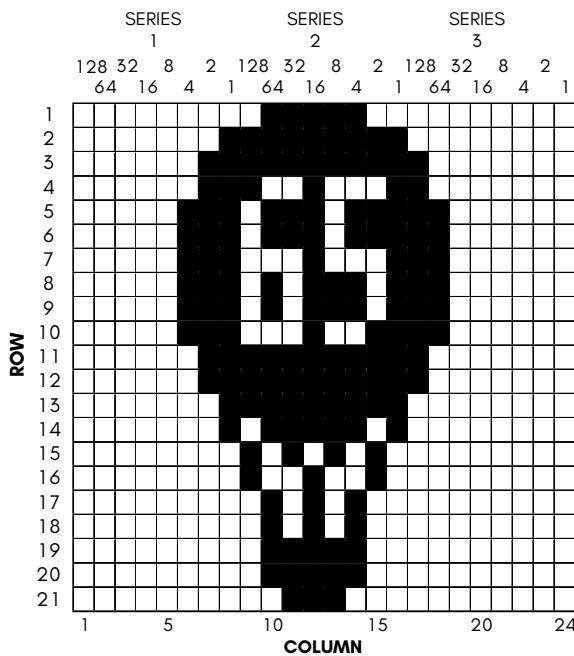
```
10 OPEN 1,8,0,"$0:*;P,R  
20 : IF DS THEN PRINT DS$: GOTO 100  
30 GET#1,X$,X$  
40 DO  
50 : GET#1,X$,X$: IF ST THEN EXIT  
60 : GET#1,BL$,BH$  
70 : LINE INPUT#1, F$  
80 : PRINT LEFT$(F$,18)  
90 LOOP  
100 CLOSE 1  
  
RUN
```

Use the "screentext" macro to perform inline screen text: ?SYNTAX ERROR

# SCREEN FONT MAPPING

# SPRITE GRIDS

# Balloon Sprite Demo



## Multi-Colour Sprite

