

MEGA 65

COMPLETE COMPENDIUM

— THE MEGA65 BOOK —



MEGA

MUSEUM OF ELECTRONIC GAMES & ART

MEGA65 TEAM

Dr. Paul Gardner-Stephen

(highlander)

Founder

Software and Virtual Hardware Architect

Spokesman and Lead Scientist

Detlef Hastik

(deft)

Co-Founder

General Manager

Marketing & Sales

Martin Streit

(seriously)

Video and Photo Production

Tax and Organization

Social Media

Anton Schneider-Michallek

(adtbm)

Hardware Pool Management

Soft-, Hard- and V-Hardware Testing

Forum Administration

Falk Rehwagen

(bluewaysw)

Jenkins Build Automation

GEOS, Hardware Quality Management

Antti Lukats

(antti-brain)

Host Hardware Design and Production

Dieter Penner

(doubleflash)

Host Hardware Review and Testing

File Hosting

Dr. Edilbert Kirk

(Bit Shifter)

MEGA65.ROM

Manual and Tools

Gábor Lénárt

(LGB)

Emulator

Mirko H.

(sy2002)

Additional Hardware and Platforms

Farai Aschwanden

(Tayger)

File Base, Tools

Financial Advisory

Gürçe Işıkyıldız

(gurce)

Tools and Enhancements

Sound

Oliver Graf

(lydon)

VHDL, Manual and Tests

Daniel England

(Mew Pokéémon)

Additional Code and Tools

Roman Standzikowski

(FeralChild)

Open ROMs

Hernán Di Pietro

(indiocolifa)

Additional Emulation

Reporting Errors and Omissions

This book is being continuously refined and improved upon by the MEGA65 community. The version of this edition is:

```
commit cddf66b97f5f02bd7b2373ed104854ff5fba3324
date: Mon Nov 7 20:00:30 2022 +1030
```

We want this book to be the best that it possibly can. So if you see any errors, find anything that is missing, or would like more information, please report them using the MEGA65 User's 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 versions of our suite of books from these locations:

- <https://mega65.org/mega65-book>
- <https://mega65.org/user-guide>
- <https://mega65.org/developer-guide>
- <https://mega65.org/basic65-ref>
- <https://mega65.org/chipset-ref>
- <https://files.mega65.org/manuals-upload>

MEGA65 REFERENCE GUIDE

Published by
the MEGA Museum of Electronic Games & Art e.V., Germany.

WORK IN PROGRESS

Copyright ©2019 – 2021 by Paul Gardner-Stephen, the MEGA Museum of Electronic Games & Art e.V., 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 reference 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.

November 7, 2022

Contents

I PREFACE	xxv
1 Introduction	xxvii
Welcome to the MEGA65!	xxix
Other Books in this series	xxx
Come Join Us!	xxx
II GETTING TO KNOW YOUR MEGA65	1-1
2 Setup	2-1
Unpacking and Connecting the MEGA65	2-3
Rear Connections	2-4
Side Connections	2-5
MEGA65 Screen and Peripherals	2-6
Optional Connections	2-7
Replacing the Real-Time Clock Battery	2-7
Operation	2-8
Using the MEGA65	2-8
The Cursor	2-9

3 Getting Started	3-1
Keyboard	3-3
Command Keys	3-3
Function Keys	3-6
The Screen Editor	3-7
Editor Functionality	3-10
4 Configuring your MEGA65	4-1
Important Note	4-3
Formatting SD cards	4-3
Installing ROM and Other Support Files	4-6
ROM File	4-7
Support Files	4-8
On-boarding	4-9
Configuration Utility	4-12
Input Devices	4-15
Chipset	4-16
Video	4-17
Audio	4-18
Network	4-19
5 Cores and Flashing	5-1
What are Cores, and Why Do They Matter?	5-5
Model types	5-5
Bitstream files	5-6
File types	5-6
Where to Download	5-6
Selecting a Core	5-7
Installing an Upgrade Core for the MEGA65	5-9

Installing Other Cores	5-11
Creating Cores for the MEGA65	5-11
Replacing the Factory Core in Slot 0	5-12
Understanding The Core Booting Process	5-12
DIP Switches	5-14
6 Floppy Disks, the Freezer, and D81 Images	6-1
Disk Drives	6-3
Disk Drive Terminology	6-3
The Freezer	6-4
Using D81 images	6-6
Auto Booting From a Disk	6-7
Saving D81 file settings	6-8
III FIRST STEPS IN CODING	6-9
7 How Computers Work	7-1
Computers are stupid. Really stupid	7-3
Making an Egg Cup Computer	7-3
8 Getting Started in BASIC	8-1
Your first BASIC programs	8-3
Exercises to try	8-12
First steps with text and numbers	8-13
Exercises to try	8-29
Making simple decisions	8-30
Exercises to try	8-43
Random numbers and chance	8-44
Exercises to try	8-48

9 Text Processing	9-1
Characters and Strings	9-3
String Literals	9-5
String Variables	9-5
String Statements	9-7
Simple Formatting	9-8
Suppressing New Lines	9-8
Automatic Tab Stops	9-8
Tabs Stops and Spacing	9-9
Sample Programs	9-9
Palindromes	9-9
Simple Ciphers	9-9
10 C64, C65 and MEGA65 Modes	10-1
Switching Modes from BASIC	10-3
From MEGA65/C65 to C64-mode	10-3
From C64 to MEGA65/C65-mode	10-4
Entering Machine Code Monitor Mode	10-4
The KEY Register	10-4
Exposing Extra C65 Registers	10-5
Disabling the C65/MEGA65 Extra Registers	10-6
Enabling MEGA65 Extra Registers	10-6
Traps to Look Out For	10-7
Accessing Memory from BASIC 65	10-7
The MAP Instruction	10-8
IV SOUND AND GRAPHICS	10-9
11 Graphics	11-1

12 Using Nexys4 boards as a MEGA65	12-1
Building your own MEGA65 Compatible Computer	12-5
Working Nexys4 Boards	12-6
The Nexys4 board	12-6
The Nexys4DDR board	12-6
The Nexys A7	12-7
Power, Jumpers, Switches and Buttons	12-8
Micro-USB Power	12-9
External Power Supply	12-9
Other Jumpers and Switches	12-10
Connections and Peripherals	12-11
Communicating with your PC	12-11
Onboard buttons	12-12
Keyboard	12-13
Some key mappings with a USB keyboard	12-14
Preparing microSDHC card	12-15
Preparation Steps	12-15
Loading the bitstream from QSPI	12-17
Preparation Steps	12-17
Widget Board	12-17
PMOD-to-Joystick Adaptor	12-20
Useful Tips	12-21

Using The Xmega65 Emulator	13-3
--------------------------------------	------

Using the Live ISO image	13-3
Creating a Bootable USB stick or DVD	13-3
Getting Started	13-4
Other Features of the Live ISO	13-5
14 Data Transfer and Debugging Tools	14-1
m65 command line tool	14-3
Screenshots using m65 tool	14-3
Load and run a program on the MEGA65	14-4
Reconfigure the FPGA to run a different bitstream	14-4
Remote keyboard entry	14-4
Unit testing and logging support	14-5
Using unit tests with C	14-5
Using unit tests with BASIC 65	14-6
BASIC 65 example	14-7
M65Connect	14-8
mega65_ftp	14-9
TFTP Server	14-10
Converting a BASIC text file listing into a PRG file	14-10
15 Assemblers	15-1
16 C and C-Like Compilers	16-1
MEGA65 libc	16-3
17 MEGA65 Standard C Library	17-1
Structure and Usage	17-3
conio.h	17-3
conionit	17-3
setscreenaddr	17-3

getscreenaddr	17-4
setcolramoffset	17-4
getcolramoffset	17-4
setcharsetaddr	17-4
getcharsetaddr	17-5
clrscr	17-5
getscreensize	17-5
setscreensize	17-5
set16bitcharmode	17-6
sethotregs	17-6
setextendedattr	17-6
togglecase	17-6
togglecase	17-6
togglecase	17-7
bordercolor	17-7
bgcolor	17-7
textcolor	17-7
revers	17-7
highlight	17-8
blink	17-8
underline	17-8
altpal	17-8
clearattr	17-9
cellcolor	17-9
setpalbank	17-9
setpalbanks	17-9
getpalbank	17-10
getpalbanks	17-10

setmapedpal	17-10
getmapedpal	17-10
setpalentry	17-10
fillrect	17-11
box	17-11
hline	17-12
vline	17-12
gohome	17-12
gotoxy	17-12
gotox	17-13
gotoy	17-13
moveup	17-13
movedown	17-14
moveleft	17-14
moveright	17-14
wherex	17-14
wherey	17-15
pcputc	17-15
pcputsxys	17-15
cputcxy	17-15
pcputs	17-16
cputc	17-16
cputnc	17-16
cputhex	17-16
cputdec	17-17
cputs	17-17
cputsxys	17-17
cputcxy	17-17

cputncxy	17-18
cprintf	17-18
pcprintf	17-19
cgetc	17-19
kbhit	17-19
getkeymodstate	17-20
flushkeybuf	17-20
cinput	17-20
VIC_BASE	17-21
18 BASIC Tokenisers	18-1
 VII APPENDICES	
APPENDICES	A-1
A Accessories	A-1
B BASIC 65 Command Reference	B-1
Commands, Functions and Operators	B-3
BASIC Command Reference	B-17
C Special Keyboard Controls and Sequences	C-1
PETSCII Codes and CHR\$	C-3
Control codes	C-5
Shifted codes	C-8
Escape Sequences	C-9

D The MEGA65 Keyboard	D-1
Hardware Accelerated Keyboard Scanning	D-3
Key Combination Tables	D-4
Unicode Basic-Latin Keyboard Map	D-10
Keyboard Theory of Operation	D-15
C65 Keyboard Matrix	D-15
Synthetic Key Events	D-16
Keyboard LED Control	D-17
Native Keyboard Matrix	D-18
E Decimal, Binary and Hexadecimal	E-1
Numbers	E-3
Notations and Bases	E-4
Decimal	E-6
Binary	E-7
Hexadecimal	E-9
Operations	E-11
Counting	E-11
Arithmetic	E-13
Logic Gates	E-15
Signed and Unsigned Numbers	E-17
Bit-wise Logical Operators	E-18
Converting Numbers	E-20
F System Memory Map	F-1
Introduction	F-3
MEGA65 Native Memory Map	F-4
The First Sixteen 64KB Banks	F-4
Colour RAM	F-5

Additional RAM	F-6
28-bit Address Space	F-6
\$D000 - \$DFFF I/O Personalities	F-8
CPU Memory Banking	F-10
C64/C65 ROM Emulation	F-11
C65 Compatibility ROM Layout	F-12
G 45GS02 Microprocessor	G-1
Introduction	G-3
Differences to the 6502	G-3
Supervisor/Hypervisor Privileged Mode	G-3
6502 Unintended Instructions	G-4
Read-Modify-Write Instruction Bug Compatibility	G-4
Variable CPU Speed	G-5
Slow (1MHz - 3.5MHz) Operation	G-5
Full Speed (40MHz) Instruction Timing	G-6
Direct Memory Access (DMA)	G-6
Accessing memory between the 64KB and 1MB points	G-6
C64-Style Memory Banking	G-6
VIC-III "ROM" Banking	G-7
VIC-III Display Address Translator	G-7
The MAP instruction	G-8
Direct Memory Access (DMA) Controller	G-10
Flat Memory Access	G-10
Accessing memory beyond the 1MB point	G-11
Using the MAP instruction to access >1MB	G-11
Flat-Memory Access	G-13
Virtual 32-bit Register	G-14
C64 CPU Memory Mapped Registers	G-16

New CPU Memory Mapped Registers	G-17
MEGA65 CPU Maths Acceleration Registers	G-19
MEGA65 Hypervisor Mode	G-24
Reset	G-24
Entering / Exiting Hypervisor Mode	G-25
Hypervisor Memory Layout	G-26
Hypervisor Virtualisation Control Registers	G-29
Programming for Hypervisor Mode	G-31
H 45GS02 & 6502 Instruction Sets	H-1
Introduction	H-3
Stack Operations	H-3
Addressing Modes	H-4
Implied	H-4
Accumulator	H-4
Q Pseudo Register	H-4
Immediate Mode	H-4
Immediate Word Mode	H-5
Base-Page Mode	H-5
Base-Page Quad Mode	H-6
Base-Page X-Indexed Mode	H-6
Base-Page Quad X-Indexed Mode	H-6
Base-Page Y-Indexed Mode	H-7
Absolute Mode	H-7
Absolute Quad Mode	H-8
Absolute X-Indexed Mode	H-8
Absolute Quad X-Indexed Mode	H-8
Absolute Y-Indexed Mode	H-9
Absolute Indirect Mode	H-9

Absolute Indirect X-Indexed Mode	H-9
Base-Page Indirect X-Indexed Mode	H-10
Base-Page Indirect Y-Indexed Mode	H-10
Base-Page Indirect Z-Indexed Mode	H-11
Base-Page Quad Indirect Z-Indexed Mode	H-11
32-bit Base-Page Indirect Z-Indexed Mode	H-12
32-bit Base-Page (Zero-Page) Indirect Quad Z-Indexed Mode	H-12
Stack Relative Indirect, Y-Indexed	H-13
Relative Addressing Mode	H-13
Relative Word Addressing Mode	H-14
6502 Instruction Set	H-15
Official And Unintended Instructions	H-15
Opcode Table	H-15
4510 Instruction Set	H-60
Instruction Timing	H-60
Opcode Table	H-60
45GS02 Compound Instructions	H-125

I Developing System Programmes	I-1
Introduction	I-5
Flash Menu	I-5
Format/FDISK Utility	I-6
Keyboard Test Utility	I-7
MEGA65 Configuration Utility	I-7
Freeze Menu	I-7
Freeze Menu Helper Programmes	I-7
Hypervisor	I-8
OpenROM	I-8

J MEGA65 Hypo Services	J-1
Introduction	J-3
General Services	J-7
Drive/Storage Services	J-11
Disk Image Services	J-40
Task and Process Services	J-44
System Partition Services	J-60
Freezer Services	J-61
K Machine Language Monitor	K-1
Introduction	K-3
The MEGA65 Machine Language Monitor	K-3
Numbers	K-4
The Assembler	K-5
Differences from the C65 Monitor	K-5
Table of MEGA65 Monitor Commands	K-6
A : ASSEMBLE	K-6
B : BITMAPS	K-8
C : COMPARE	K-9
D : DISASSEMBLE	K-9
F : FILL	K-10
G : GO	K-11
H : HUNT	K-11
J : JUMP	K-11
L : LOAD	K-11
M : MEMORY	K-12
R : REGISTERS	K-13
S : SAVE	K-13
T : TRANSFER	K-13

V : VERIFY	K-13
X : EXIT	K-14
. : ASSEMBLE	K-14
> : MODIFY MEMORY	K-14
; : MODIFY REGISTERS	K-14
@ : DISK COMMAND	K-15
The Matrix Mode/Serial Monitor	K-15
Table of Matrix Mode Monitor Commands	K-17
Calling the Monitor	K-17
# : Hypervisor trap enable/disable	K-17
+ : Set Serial Interface UART Divisor	K-18
@ : CPUMEMORY	K-18
? or H : HELP	K-19
B : BREAKPOINT	K-19
D : DISASSEMBLE	K-19
E : FLAGWATCH	K-20
F : FILL	K-20
G : SETPC	K-20
I : INTERRUPTS	K-21
J : DEBUGMON	K-21
L : LOADMEMORY	K-21
M : MEMORY	K-21
R : REGISTERS	K-22
S : SETMEMORY	K-22
T : TRACE	K-22
W : WATCHPOINT	K-22
Z : CPUHISTORY	K-23

L F018-Compatible Direct Memory Access (DMA) Controller	L-1
F018A/B DMA Jobs	L-5
F018 DMA Job List Format	L-6
F018 11 byte DMA List Structure	L-7
F018B 12 byte DMA List Structure	L-7
Performing Simple DMA Operations	L-9
MEGA65 Enhanced DMA Jobs	L-14
Texture Scaling and Line Drawing	L-17
Inline DMA Lists	L-19
Audio DMA	L-20
Sample Address Management	L-21
Sample Playback frequency and Volume	L-21
Pure Sine Wave	L-22
Sample playback control	L-22
F018 “DMAgic” DMA Controller	L-23
MEGA65 DMA Controller Extensions	L-23
Unimplemented Functionality	L-27
M VIC-IV Video Interface Controller	M-1
Features	M-5
VIC-II/III/IV Register Access Control	M-6
Detecting VIC-II/III/IV	M-7
Video Output Formats, Timing and Compatibility	M-8
Integrated Marvellous Digital Hookup™ (IMDH™) Digital Video Output . M-9	M-9
Connecting to Naughty Proprietary Digital Video Standards . M-9	M-9
Frame Timing	M-11
Physical and Logical Rasters	M-14
Bad Lines	M-14
Memory Interface	M-15

Relocating Screen Memory	M-15
Relocating Character Generator Data	M-16
Relocating Colour / Attribute RAM	M-16
Relocating Sprite Pointers and Images	M-17
Hot Registers	M-18
New Modes	M-20
Why the new VIC-IV modes are Character and Bitmap modes, not Bit-plane modes	M-20
Displaying more than 256 unique characters via "Super-Extended Attribute Mode"	M-21
Default Bit Fields (when GOTOX bit is cleared):	M-22
Bit Fields when GOTOX bit is set:	M-23
Using Super-Extended Attribute Mode	M-25
Full-Colour (256 colours per character) Text Mode (FCM)	M-29
Nibble-colour (16 colours per character) Text Mode (NCM)	M-29
Alpha-Blending / Anti-Aliasing	M-30
Flipping Characters	M-30
Variable Width Fonts	M-30
Raster Re-write Buffer	M-31
Sprites	M-31
VIC-II/III Sprite Control	M-32
Extended Sprite Image Sets	M-32
Variable Sprite Size	M-32
Variable Sprite Resolution	M-33
Sprite Palette Bank	M-33
Full-Colour Sprite Mode	M-34
VIC-II / C64 Registers	M-37
VIC-III / C65 Registers	M-39
VIC-IV / MEGA65 Specific Registers	M-42

N Sound Interface Device (SID)	N-1
SID Registers	N-3
O 6526 Complex Interface Adaptor (CIA) Registers	O-1
CIA 6526 Registers	O-3
CIA 6526 Hypervisor Registers	O-6
P 4551 UART, GPIO and Utility Controller	P-1
C65 6551 UART Registers	P-3
4551 General Purpose I/O & Miscellaneous Interface Registers	P-4
Q 45E100 Fast Ethernet Controller	Q-1
Overview	Q-3
Differences to the RR-NET and similar solutions	Q-3
Theory of Operation: Receiving Frames	Q-4
Accessing the Ethernet Frame Buffers	Q-6
Theory of Operation: Sending Frames	Q-7
Advanced Features	Q-7
Broadcast and Multicast Traffic and Promiscuous Mode	Q-8
Debugging and Diagnosis Features	Q-8
Memory Mapped Registers	Q-9
COMMAND register values	Q-10
Example Programs	Q-11
R 45IO27 Multi-Function I/O Controller	R-1
Overview	R-3
F011-compatible Floppy Controller	R-3
Multiple Drive Support	R-3
Buffered Sector Operations	R-4
Reading Sectors from a Disk	R-4

Track Auto-Tune Function Deprecated	R-5
Sector Skew and Target Any Mode	R-5
Disk Layout and 1581 Logical Sectors	R-6
FD2000 Disks	R-7
High-Density and Variable-Density Disks	R-7
Track Information Blocks	R-8
Formatting Disks	R-9
Write Pre-Compensation	R-10
Buffered Sector Writing	R-10
F011 Floppy Controller Registers	R-11
SD card Controller and F011 Virtualisation Functions	R-13
SD card Based Disk Image Access	R-14
F011 Virtualisation	R-16
Dual-Bus SD card Controller	R-17
Write Gate	R-17
Fill Mode	R-17
Selecting Among Multiple SD cards	R-18
SD Controller Command Table	R-18
Touch Panel Interface	R-20
Audio Support Functions	R-23
Miscellaneous I/O Functions	R-26
S Reference Tables	S-1
Units of Storage	S-3
Base Conversion	S-4
T Flashing the FPGAs and CPLDs in the MEGA65	T-1
Suggested PC specifications	T-5
Warning	T-6

Installing Vivado	T-6
Installing the FTDI drivers	T-22
Linux drivers	T-23
Windows drivers	T-23
Flashing the main FPGA using Vivado	T-27
Flashing the CPLD in the MEGA65's Keyboard with Lattice Diamond	T-40
Flashing the MAX10 FPGA on the MEGA65's Mainboard with INTEL QUARTUS	T-47
U Trouble shooting	U-1
Hardware	U-3
No lights when powering on	U-3
Vivado	U-3
RAM requirements	U-3
mega65_ftp	U-3
Missing Library	U-4
V Model Specific Features	V-1
Detecting MEGA65 Models	V-3
MEGA65 Desktop Computer, Revision 3 onwards	V-3
MEGA65 Desktop Computer, Revision 2	V-4
MEGAphone Handheld, Revisions 1 and 2	V-5
Nexys4 DDR FPGA Board	V-6
W Schematics	W-1
MEGA65 R3 Schematics	W-3
MEGA65 R2 Schematics	W-28
Nexys Widget Board Schematics	W-51

X	Supporters & Donors	X-1
Organisations	X-3
Contributors	X-4
Supporters	X-5

INDEX	Index-3
--------------	----------------

PART I

PREFACE

1

CHAPTER

Introduction

- Welcome to the MEGA65!
- Other Books in this series
- Come Join Us!

WELCOME TO THE MEGA65!

Congratulations on your purchase of one of the most long-awaited computers in the history of computing! The MEGA65 is community designed, and based on the never-released Commodore® 65¹ computer; a computer designed in 1989 and intended for public release in 1990. Decades have passed, and we have endeavoured to invoke memories of an earlier time when computers were simple and friendly. They were not only simple to operate and understand, but friendly and approachable for new users.

These 1980s computers inspired many of their owners to pursue the exciting and rewarding technology careers they have today. Just imagine the exhilaration these early computing pioneers experienced, as they learned they could use their new computer to solve problems, write a letter, prepare taxes, invent new things, discover how the universe works, and perhaps even play an exciting game or two! We want to re-awaken that same level of excitement (which alas, is no longer found in modern computing), so we have created the **MEGA65**.

The MEGA65 team believes that owning a computer is like owning a home. You don't just use a home; you change things, big and small, to make it your own custom living space. After a while, when you settle in, you may decide to renovate or expand your home to make it more comfortable, or provide more utility. Think of the MEGA65 as your very own "computing home".

This guide will teach you how to do more than just hang pictures on a wall; it will show you how to build your dream home. While you read this user's guide, you will learn how to operate the MEGA65, write programs, add additional software, and extend hardware capabilities. What won't be immediately obvious is that along the journey, you will also learn about the history of computing as you explore the many facets of BASIC version 65 and operating system commands.

Computer graphics and music make computing more fun, and we designed the MEGA65 to be fun! In this user's guide, you will learn how to write programs using the MEGA65's built-in **graphics** and **sound** capabilities. But you don't need to be a programmer to have fun with the MEGA65. Because the MEGA65 includes a complete Commodore® 64™², it can also run thousands of existing games, utilities, and business software packages, as well as new programs being written today by Commodore computer enthusiasts. Excitement for the MEGA65 will grow as we all witness the programming marvels our MEGA65 community create, as they (and you!) discover and master the powerful capabilities of this modern Commodore computer recreation. Together, we can build a new "homebrew" community, teeming with software

¹Commodore is a trademark of C= Holdings

²Commodore 64 is a trademark of C= Holdings

and projects that push the MEGA65's capabilities far beyond what anyone thought would be possible.

We welcome you on this journey! Thank you for becoming a part of the MEGA65 community of users, programmers, and enthusiasts!

OTHER BOOKS IN THIS SERIES

This book is one of several within the MEGA65 documentation suite. The series includes:

- **The MEGA65 User's Guide**

Provides an introduction to the MEGA65, and a condensed BASIC 65 command reference

- **The MEGA65 BASIC 65 Reference**

Comprehensive documentation of all BASIC 65 commands, functions and operators

- **The MEGA65 Chipset Reference**

Detailed documentation about the MEGA65 and C65's custom chips

- **The MEGA65 Developer's Guide**

Information for developers who wish to write programs for the MEGA65

- **The MEGA65 Complete Compendium**

(Also known as **The MEGA65 Book**)

All volumes in a single huge PDF for easy searching. 1080 pages and growing!

COME JOIN US!

Get involved, learn more about your MEGA65, and join us online at:

- <https://mega65.org/chat>
- <https://mega65.org/forum>

PART II

**GETTING TO KNOW YOUR
MEGA65**

CHAPTER 2

Setup

- **Unpacking and Connecting the MEGA65**
- **Rear Connections**
- **Side Connections**
- **MEGA65 Screen and Peripherals**
- **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:

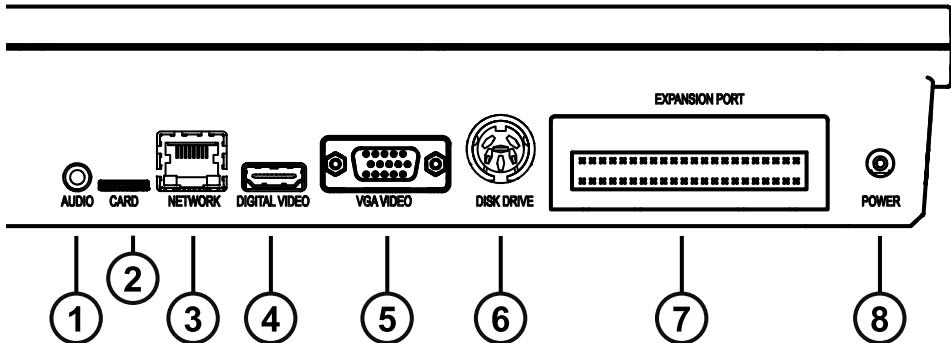
- A television or computer monitor with a VGA or digital video input, that is capable of displaying an image at 480p or 576p (720x480 or 720x576 pixel resolution at 50Hz or 60Hz)
- A VGA video cable, or;
- A digital video cable

These items are not included with the MEGA65.

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

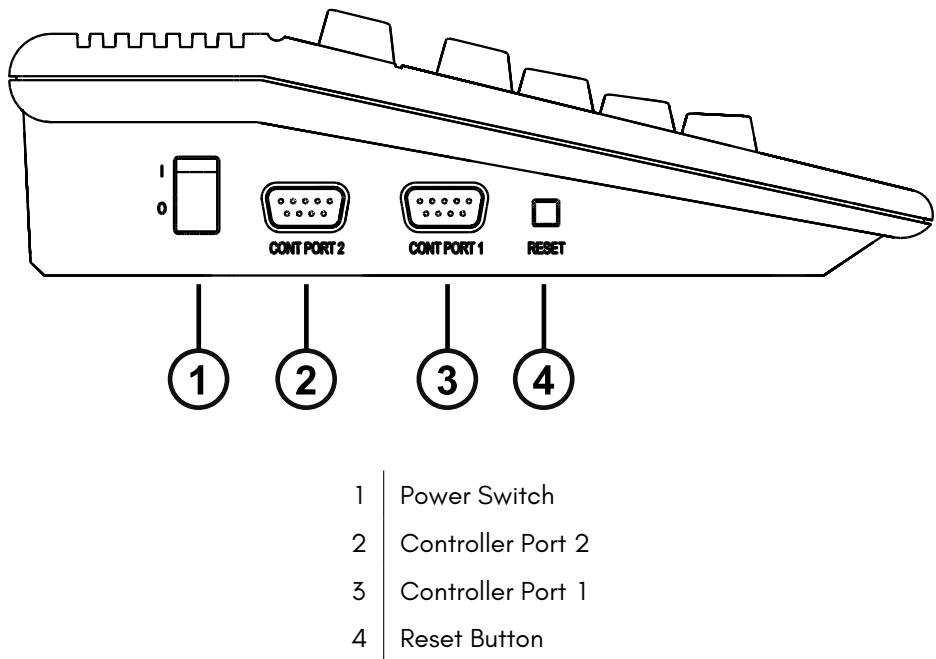
- 3.5mm mini-jack audio cable and suitable speakers or hi-fi system, so that you can enjoy the sound capabilities of your MEGA65.
- RJ45 Ethernet cable (regular network cable) and a network router or switch. This allows the usage of the high-speed networking capabilities of your MEGA65.

REAR CONNECTIONS



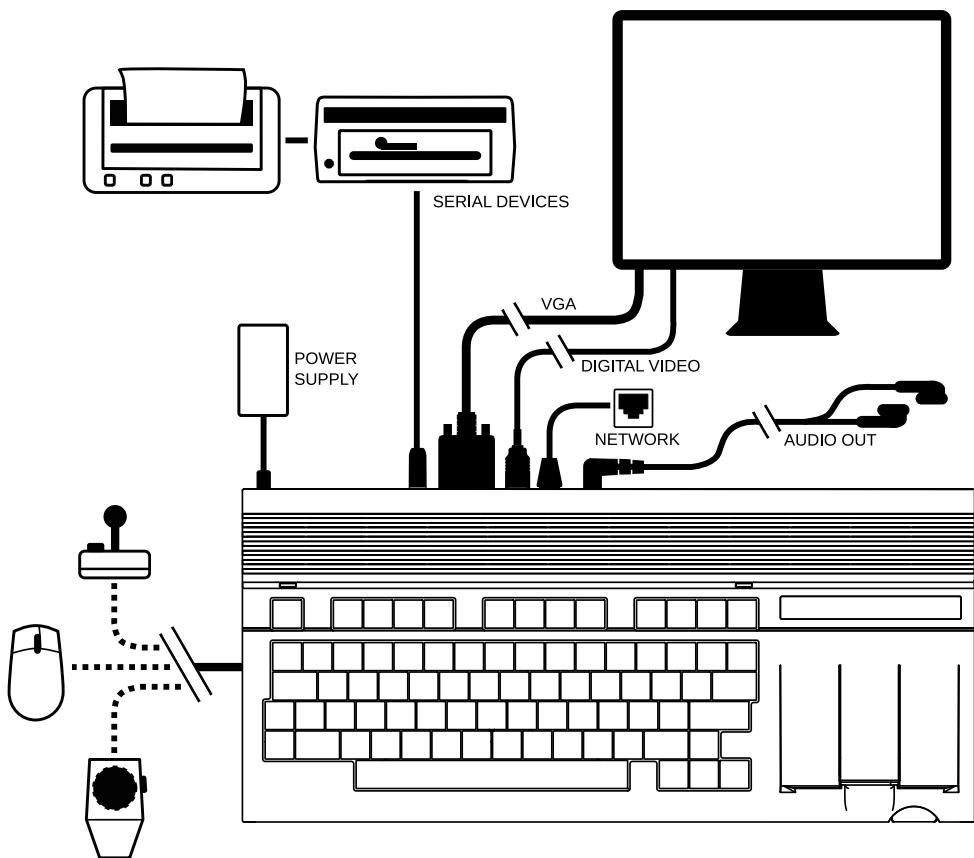
- | | |
|---|---|
| 1 | 3.5mm Audio Mini-Jack |
| 2 | External microSD Card Slot |
| 3 | Network LAN Port |
| 4 | Digital Video Connector (including sound) |
| 5 | VGA Video Connector |
| 6 | IEC Serial Bus Connector for Disk Drives and Printers |
| 7 | Cartridge Expansion Port |
| 8 | Power Supply Socket |

SIDE CONNECTIONS



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

MEGA65 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 use a Digital Video to DVI adapter.

OPTIONAL CONNECTIONS

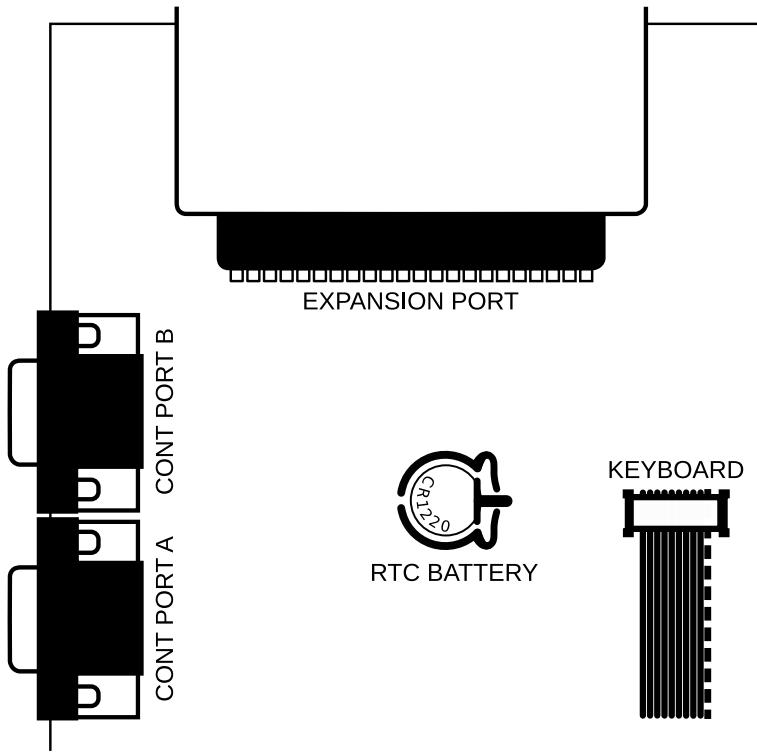
1. The MEGA65 includes an internal 3.5" floppy disk drive. You can also connect older Commodore® IEC serial floppy drives to the MEGA65, such as the Commodore 1541, 1571 or 1581. To use these drives, connect one end of an IEC cable to the Commodore floppy disk drive and the other end to the Disk Drive socket of the MEGA65. You can also connect SD2IEC devices and Pi1541's. It is also possible to daisy-chain additional floppy disk drives or Commodore compatible printers.
2. You can connect your MEGA65 to an Ethernet 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 a 3.5mm mini-jack to twin RCA adapter cable. The MEGA65 also has a built-in amplifier to allow the use of headphones.
4. A microSD card (either SDHC or SDXC) can be inserted into the external microSD card slot at the rear of the MEGA65.
5. Underneath the MEGA65, you will find an opening/trapdoor that provides access to the internal SD card slot, and also two PMOD connectors that allow for future possible hardware expansions such as Tape adapters, Userport interfaces, extra memory, or even real SIDs.

Replacing the Real-Time Clock Battery

The MEGA65 includes a Real-Time Clock, which is used to display the time and date on the startup screen, to add timestamps to files that the MEGA65 writes to your SD cards, and by the **DT\$** and **TI\$** BASIC65 commands. This clock utilises a CR1220 coin-cell battery to keep time when the MEGA65 isn't switched on, but unfortunately the MEGA65 doesn't ship with one (this is to minimise the chance of any international shipping related issues).

To change the battery, you will need to open the case, exposing the motherboard. The case is held together with three screws, all of which are along the bottom of the front side of the case. Once the three screws have been removed, **carefully** take the top half of the case off, and note the orientation of the keyboard connector, before disconnecting it.

The battery is located between the controller ports and the keyboard connector:



If you are removing an existing battery, push the battery release lever on the bottom (flat-sided) side of the battery socket away from the battery to remove it. Next, insert the new battery with the side labelled + **facing up**, and press it into place.

Once you have re-assembled your MEGA65, you can set the time in the Configure menu. For more information on how to set the Real-Time Clock, refer to the Configuration Utility section on page [4-12](#).

OPERATION

Using the MEGA65

1. Switch on the MEGA65 by using the switch on the left-hand side.

Note: On first power up, you will see an On-Boarding screen. See page [4-9](#) for more details.

Welcome to the MEGA65!

Before you go further, there are couple of things you need to do.

Press F3 - F13 to set the time and date.

Time: 23:36:11 04/Nov/2021
F3 F5 F7 F9 F11 F13

Video: DVI (no sound), NTSC 60Hz
TAB = cycle through modes

- After a moment, the following will be displayed on your TV or monitor:



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 their respective characters onto the screen. The characters will be printed at the current cursor position, and the cursor will advance to the next position after every key press.

Here you can type commands, that can do things such as loading a program. You can also start entering program code!

CHAPTER

3

Getting Started

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

KEYBOARD

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

You may notice that the keyboard is a little different from the keyboards 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:  ,  ,  ,  , and  .

RETURN

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

SHIFT

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

In upper case mode, holding down  and pressing any key with two graphic symbols on the front produces the right-hand symbol on that key. For example,  and  prints the ☒ character.

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

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

SHIFT LOCK

In addition to  is  . Press this key to lock down the Shift function. Now any key you press while  is illuminated prints the character to the screen as if you were holding down  . This 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 (from **1** to **8**) allows you to change text colours. The colour that is printed at the top row on the front of the number key will be used. Holding down **CTRL** and pressing **9** or **0** switches reverse-text mode on and off.

There are some examples of this on page [3-7](#), and all of the Control Functions are listed on page [C-5](#).

If a program is being **LISTed** to the screen, holding down **CTRL** slows down the display of each line. You can read more about the **LIST** command on page [B-145](#).

Holding **CTRL** and pressing ***** enters the Matrix Mode Debugger (refer to the **MEGA65 Book** for more details).

RUN STOP

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

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

You can boot your MEGA65 into the **Machine Code Monitor** by holding down **RUN STOP** and pressing reset on the left-hand side.

RESTORE

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

You can also enter the **Freezer** by pressing and holding **RESTORE** for half to one second. From there you can access the Machine Code Monitor. You can read more about the Freezer on page [6-4](#).

THE CURSOR KEYS

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

The cursor moves in the direction indicated on the keys: .

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

You don't have to keep pressing a cursor key over and over. If you need to move the cursor a long way, you can keep the key pressed down. When you are finished, simply release the key.

ARROW KEYS

These keys are different to the cursor keys! They are (next to) and (next to). Both arrow keys are used in various BASIC functions and escape sequences.

For example, can be used as a shortcut for **SAVE**, and is used to raise a number to a power (which is the same as multiplying a number by itself a specified number of times).

You can read more about the available escape sequences on page [C-9](#).

These two PETSCII specific keys will always be shown in MEGA65 literature with a white background.

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 and press . All the characters to the right of the cursor are shifted to the right. This allows you to type a letter, number or any other character at the newly inserted space.

CLeaR/HOME

Pressing places the cursor at the top left-most position of the screen.

Holding down and pressing clears the entire screen and places the cursor at the top left-most position of the screen.

MEGA KEY

 or the MEGA key provides a number of different functions and can be used to launch special utilities.

Holding  and pressing  switches between lower and uppercase character modes.

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

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, i.e., the colour that is printed at the bottom row on the front of the number key will be used.

Holding  and pressing  enters the Matrix Mode Debugger (refer to the **MEGA65 Book** for more details).

Switching on the MEGA65 or pressing the reset button on the left-hand 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. This feature is not available in C64-mode.

Function Keys

There are seven Function keys available for use by software applications,       and  can be used to perform specific 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

 can be used by software and also acts as  / .

ALT

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

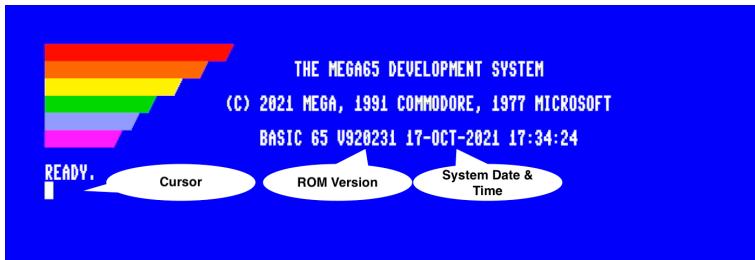
Holding **ALT** down while switching the MEGA65 on activates the Utility Menu. You can format an SD card, or enter the MEGA65 Configuration Utility to select the default video mode and change other settings, or to test your keyboard.

CAPS LOCK

CAPS LOCK works similarly to **SHIFT LOCK** in C65 and MEGA65-modes, but only modifies the letter keys. Also, holding **CAPS LOCK** down forces the processor to run at the maximum speed. This can be used for things such as speeding 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 fraction of a second.

THE SCREEN EDITOR

When you switch on your MEGA65 or reset it, the following screen will appear:

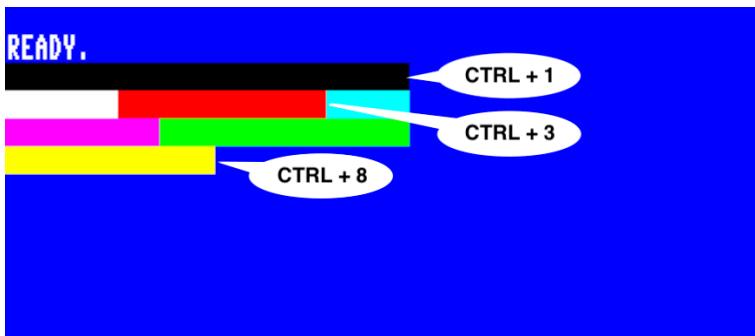


The colour bars in the top left-hand side of the screen can be used as a guide to help calibrate the colours of your display. The screen also displays the name of the system, the copyright notice, and the ROM version. The displayed date and time are taken from the internal RTC (Real-Time Clock), which can be set in the Configure Menu.

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 at the cursor position. The cursor itself will advance after each key press.

You can also produce reverse text or colour bars by holding down **CTRL** and pressing **9**, or **R**. This enters reverse text mode. When this is enabled, you can press and hold the **SPACE** bar. While doing so, a white bar will be drawn across the screen. You can even change the current colour by holding **CTRL** down and pressing a number key (from **1** to **8**). For example, if you press and hold **CTRL** down and press **1**, the colour will change to black. Now, when you hold down the **SPACE** bar, a black bar will be drawn. If you continue to change the colour and press the **SPACE** bar, you will get an effect similar to the following image:



You can disable reverse text mode by holding **CTRL** and pressing **0**.

By pressing any key, characters will be printed to the screen in the chosen colour.

A further eight colours can be selected by holding down **M** and pressing a key from **1** to **8**. The colour that is printed at the bottom row on the front of the number key will be used. For example, if you held **M** down while pressing **4**, dark grey will be used. For access to an additional 16 colours of the alternate/rainbow palette, refer to the **CTRL** + **A** shortcut described on page C-5.

Note:

- **Quote Mode:** If you were to press **"** to open a string, and then try to change colours, reverse text, move the cursor keys, or use the **CLR HOME** key, instead of these actions instantly occurring, funny PETSCII symbols will appear instead. This is due to a BASIC facility called *quote mode* (described further in the **MEGA65 Book**), which allows you to encode such actions into a string so that they can be executed at a later time (for example, via a **PRINT** statement within your programs). To end *quote mode*, simply type another **"** to mark the end of your string.

- **Insert Mode:** A similar facility is called *insert mode*, where for the number of times you press **SHIFT** + **INST DEL** to insert a few spaces, the same number of key-presses that follow it will abide by the same principles of *quote mode*.
 - You can forcefully exit either of these modes by pressing **ESC**, **O**.

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



What will you draw?

Functions

Functions using **CTRL** are called **Control Codes**. Functions using **SHIFT** are called **Mega Codes**. There are also functions that are called by using **SHIFT**, which are called **Shifted Codes**.

Lastly, **ESC** enables the use of **Escape Sequences**.

You can read about all of these functions in detail on page [C-5](#), but some are shown in this section.

ESC Sequences

Escape sequences are performed a little differently than a Control function or a Shift function. Instead of holding the modifier key down, an Escape sequence is performed by pressing **ESC** and releasing it, followed by pressing the desired key.

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

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

EDITOR FUNCTIONALITY

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

For example, press **CLR HOME** to go to the home position on the screen. Hold **CTRL** down and press **W** 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 **CLR HOME** and then **→** to move the cursor to the fourth column. Hold down **CTRL** and press **X** to set a tab. Move another 16 positions to the right, and press **CTRL** and **X** again to set a second tab.

Press **CLR HOME** to go back to the home position. Hold **CTRL** down and press **I**. 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 do you ask? 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 **CTRL** and **I** to advance to the 16th and 20th positions.

Creating a Window

You can set a window on the MEGA65 working screen. Move your cursor to the beginning of the "BASIC 65" text. Press **ESC**, then press **T**. Move the cursor 10 lines down and 15 to the right.

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

For example, if you were to type LIST to list out a program, the listing will be confined to the window region you have specified:



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

Additional ASCII characters

You may have noticed a few ASCII characters on the MEGA65 keyboard that aren't traditionally a part of the PETSCII character set. In order to make use of these from within BASIC:

- Type either **FONT A** or **FONT B**.
- Press **Y** + **SHIFT** to switch to lowercase.

You will now be able to type those additional ASCII characters via the keyboard. To revert back to the original PETSCII character set, type **FONT C**.

Extras

Here are some extra, useful things you can do whilst in the screen editor:

To enter the **Freezer**, press and hold **RESTORE** for half to one second, then press **J** to switch controller ports without having to physically swap the controller to the other port. You can read more about the Freezer, and what else it can do on page [6-4](#).

To enter **Fast mode** (40.5MHz), either type **SPEED** (preferred way), or **POKE 0,65** or enter the Freeze Menu and switch the CPU frequency with **F**.

Return back to **Slow mode** (1MHz) by either typing **SPEED 1** (preferred way), or **POKE 0,64** or enter the Freeze Menu and switch the CPU frequency with **F**.

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

CHAPTER 4

Configuring your MEGA65

- **Important Note**
- **Formatting SD cards**
- **Installing ROM and Other Support Files**
- **On-boarding**
- **Configuration Utility**

IMPORTANT NOTE

For your convenience, your MEGA65 comes with an SD card with all of the essential files already on it, so you may prefer to skip this section and jump straight to the on-boarding section on page [4-9](#).

Alternatively, you're welcome to read this section and familiarise yourself on how your SD card was prepared.

Do not format the SD card that came with your MEGA65. If you want to create a new bootable SD card, please use another one, and keep the SD card that came with your MEGA65 as a safety backup.

FORMATTING SD CARDS

The MEGA65 has two SD card slots: A full-size SD card slot inside, under 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 default to the external slot. The exception to this is that the MEGA65's FDISK/FORMAT utility can access both, allowing you to select which SD card to format or repair.

If you wish to use a different SD card to the pre-configured one supplied with your MEGA65, the new SD card must be formatted first.

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

Only use SDHC cards. 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 as to which file-system is currently on the card, as the MEGA65 FDISK/FORMAT utility will completely reformat the card.

Why must I format the SD card via the MEGA65 (and not on a PC/computer)?

There are several reasons for this: First, 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 between MEGA65 programs and games.

Formatting an SD card on the MEGA65 is easy. Switch the MEGA65 on while holding **ALT** down.

This will present the MEGA65 Utility Menu, which contains a selection of built-in utilities, similar to the following:



Note that the Utility Menu is always accessible, even if no SD card is present in both the internal and external slots.

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 (which is called **SDCARD FDISK+FORMAT UTILITY** in the screenshot above), 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 0, and the external microSD card is bus 1. Note that the MEGA65 will always attempt to boot from the external microSD card if one is present.

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 on the MEGA65 cannot be undone, and 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 them 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 back up 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 (Internal SD slot):
Maximum readable sector is $01CD9FFE
14771 MiB SD CARD FOUND.
SD Card read speed = 1422 KB/sec

Current partition table:
0C : Start=0 /0 /0      or 00000000 / End=0 /0 /0      or 018D97FE
41 : Start=0 /0 /0      or 018D9FFE / 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 (External MicroSD slot):
No card detected on bus 1

Please select SD card to modify: 0/1

MEGA65 FDISK#FORMAT V00.10 : (C) COPYRIGHT 2017-2020 PAUL GARDNER-STEPHEN ETC.
```

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

```
Please select SD card to modify: 0/1
Maximum readable sector is $01CD9FFE
14771 MiB SD CARD FOUND.
SD Card read speed = 1186 KB/sec

Current partition table:
0C : Start=0 /0 /0      or 00000000 / End=0 /0 /0      or 018D97FE
41 : Start=0 /0 /0      or 018D9FFE / 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.
$018D97FE Sectors available for VFAT32 partition.

Format Card with new partition table and FAT32 file system?
12722 MiB VFAT32 Data Partition @ 00000000;
$003198ED Clusters, 25396 Sectors/FAT, 568 Reserved Sectors.
2048 MiB MEGA65 System Partition @ 018D9FFE;

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

MEGA65 FDISK#FORMAT V00.10 : (C) COPYRIGHT 2017-2020 PAUL GARDNER-STEPHEN ETC.
```

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

It is also possible to attempt a recovery from a lost Master Boot Record error (also known as the Boot Sector), by typing **FIX MBR** instead.

The aim here is to have a correctly formatted SD card with all of the essential files stored on it so the MEGA65 can properly boot. When switching on, the MEGA65 will search for, and boot using these files:

- **FREEZER.M65** (Freeze Menu program)
- **AUDIO MIX.M65** (Freeze Menu audio mixer utility)
- **C64THUMB.M65** (C64 thumbnail image used in freezer)
- **C65THUMB.M65** (C65 thumbnail image used in freezer)
- **MEGA65.ROM** (128KB ROM file)
- **MEGA65.D81** (Disk image)

Straight out of the box, the MEGA65 will only have one SD card installed, accessible via the trap-door under the case. This SD card contains all of the essential files needed to properly boot up. If an external microSD card is also detected during boot up, the MEGA65 will give it higher priority, and will try to boot from it instead. This means that the external microSD card needs to have the essential files on it, otherwise the MEGA65 will not boot up properly and will fall back to loading the OpenROM, which does not support all MEGA65 features. In general, if your MEGA65 cannot boot properly and falls back to OpenROM, your boot-up screen will look similar to this:



INSTALLING ROM AND OTHER SUPPORT FILES

The MEGA65 FDISK/FORMAT utility will add a copy of the Open ROMs project's C64-compatible ROM to your SD card, and will name it **MEGA65.ROM**.

For MEGA65 owners, we have replaced this file with the latest ROM from the 'Closed ROMs' project. It provides many improvements over the original/incomplete C65

ROMs. It contains the operating system, BASIC 65, CBDOS and the machine language monitor BSMON. This ROM is developed especially for the MEGA65 and can be identified by the version number **92XXXX**.

However, you may have other ROMs that you wish to use on your MEGA65. You can copy as many of these as you wish onto the SD card, just make sure that they have the .ROM file extension. The default ROM should be called "MEGA65.ROM". These files must be 128KB in size, and use the same internal format as the ROMs intended for the C65. This means that the C64-mode KERNEL must be placed at offset \$E000, a C65-mode BASIC at \$A000, and a suitable character set at \$D000.

You can optionally name your alternate ROMs as MEGA65x.ROM, replacing x with a number from 0 to 9. This allows you to quickly boot-up to your alternate ROMs by holding down a number from **0** to **9** prior to switching on your MEGA65.

Other important support files include FREEZER.M65 and AUDIOMIX.M65, which allow you to use the MEGA65's integrated Freezer. More details are provided in the 'Floppy Disks, D81 Images, and the Freezer' chapter on page [6-3](#).

ROM File

Original C65 ROMs

You may want to source your own C65 ROM via other means. There were many different versions created during the development of the Commodore 65, and the MEGA65 can use any of them. However, they will not support the advanced features of the MEGA65, and are incomplete and buggy, as development on them ceased due to Commodore abandoning the C65 project.

MEGA65 Closed ROMs

There are newer versions of the MEGA65 Closed ROM under development. These ROMs improve upon the original C65 ROMs and make better use of the extra hardware capabilities that the MEGA65 has over the original C65 hardware. These ROMs are available via the filehost (<https://files.mega65.org>), but only to owners of the MEGA65, who will need to login to the filehost with their credentials in order to download it. It can be located by visiting the **Files** tab and searching for "**kernal rom**":

The screenshot shows a web browser displaying the filehost website. At the top, there is a navigation bar with tabs for "News", "Files" (which is highlighted with a pink circle), and "Articles". Below the navigation bar, there is a search bar containing the text "kernal rom" and a magnifying glass icon. A pink arrow points from the search bar to the magnifying glass icon. Another pink arrow points from the search bar to the "Files" tab. The main content area shows a table of search results. The first result is a row for "C65/MEGA65 Kernal ROM", which includes a download icon, the number 371, a rating of 5.0 (2), the category "Firmware", the type "ROM File", the OS "MEGA65, C65", the filename "920236.BIN", the size "131 KB", the author "adtbm", and the published date "2021/10/31".

Title	Download	Rating	Category	Type	OS	Filename	Size	Author	Published
C65/MEGA65 Kernal ROM		371	5.0 (2)	Firmware	ROM File	920236.BIN	131 KB	adtbm	2021/10/31

MEGA65 ROM diff files

If you have sourced your own 911001.bin C65 ROM and would like to patch it to the latest MEGA65 closed ROM, we do provide patches, as the additional improvements we have made to the closed rom are open source. These diff files are available at <http://mega65.org/rom-diffs>

MEGA65 Open ROMs

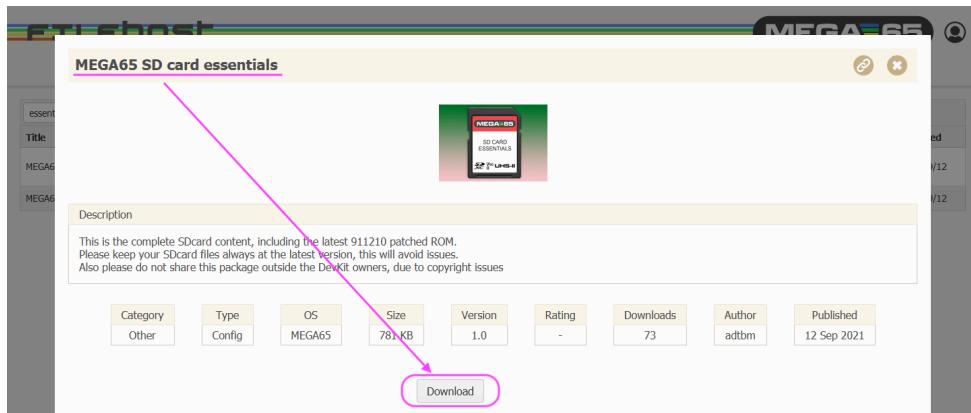
Another available option is to make use of **MEGA65 Open ROMs**. The latest version of this is always downloadable from either of the following urls:

- <http://mega65.org/open-roms>
- <https://github.com/MEGA65/open-roms/raw/master/bin/mega65.rom>

Support Files

For official owners of the MEGA65 (both the devkit and the final product), visit the following url and log in with the user credentials you have been provided. This will take you to the MEGA65 filehost location where the **MEGA65 SD card essentials** download page is located. Click the **Download** link to retrieve the latest **SD essentials.rar** file.

<http://mega65.org/sdcard-files>



Note that this link is only available to official owners of the MEGA65 product, as the fileset also contains the licensed closed-source MEGA65.ROM file. For Nexys board owners in search of a similar fileset (without the ROM), visit the following url instead: <http://mega65.org/sdcard-norom>

This will take you to the MEGA65 filehost location where the **MEGA65 SD card essentials - No ROM** download page is located. Click the **Download** link to retrieve the latest **SD essentialsNoROM.rar** file.

Note that while this fileset does not contain a ROM, there are future plans for it to include the freely available ROM from the Open ROMs project.

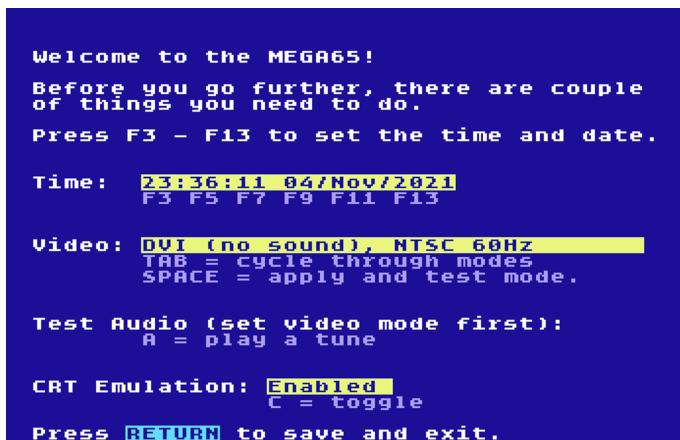
The screenshot shows a software interface with a dark header bar. In the center, there is a file listing titled "MEGA65 SD card essentials - No ROM". Below the title is a small thumbnail image of an SD card labeled "SD CARD ESSENTIALS NO ROM" and "SD 2nd UHS-II". The main area contains a table with the following data:

Category	Type	OS	Size	Version	Rating	Downloads	Author	Published
Software	SD Card Files	MEGA65	63 KB	1.0	-	266	adtmb	12/20

A pink arrow points from the title "MEGA65 SD card essentials - No ROM" down to the "Download" button at the bottom of the listing.

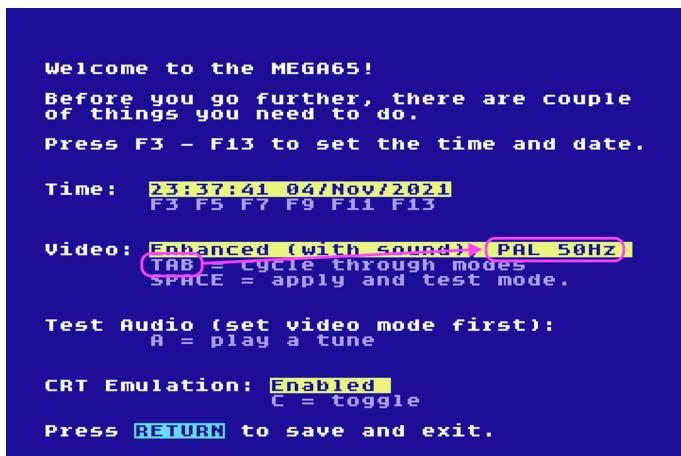
ON-BOARDING

On first launch of your MEGA65, you will see the on-boarding screen:

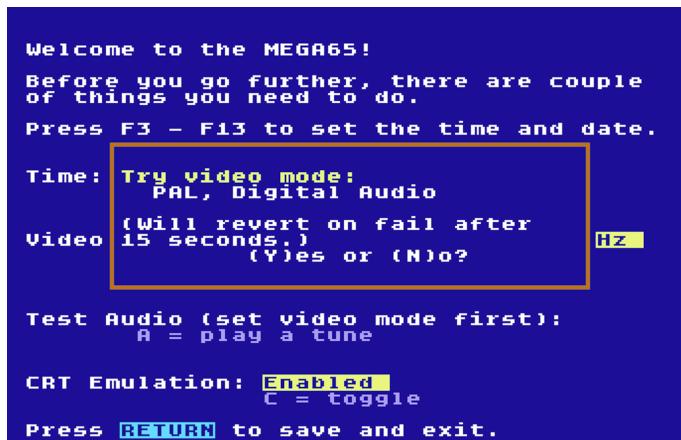


Here you can select and test your screen configuration.

For example, press **TAB** to switch to PAL 50Hz:



Then press **SPACE**, followed by **Y** to test the new video mode:



Press **K** to keep the new video mode:

```
Welcome to the MEGA65!
Before you go further, there are couple
of things you need to do.
Press F3 - F13 to set the time and date.

Time: Press K to keep video mode.
      Timeout in 13 sec.

Video [Hz] [ ]
```

Test Audio (set video mode first):
A = play a tune

CRT Emulation: Enabled
C = toggle

Press RETURN to save and exit.

Press **RETURN** to complete the configuration:

```
Welcome to the MEGA65!
Before you go further, there are couple
of things you need to do.
Press F3 - F13 to set the time and date.

Time: 23:41:07 04/Nov/2021
      F3 F5 F7 F9 F11 d13

Video: Enhanced (with sound), PAL 50Hz
      TAB = cycle through modes
      SPACE = apply and test mode.

Test Audio (set video mode first):
      A = play a tune

CRT Emulation: Enabled
      C = toggle

Press RETURN to save and exit.
```

Note for Nexys4 board users:

At this very specific step, the board is supposed to reboot and display the main MEGA65 screen. If the board does not reboot and the screen remains black, then switch power to the board off then on again.

After the MEGA65 reboots you will be greeted with the main MEGA65 screen:



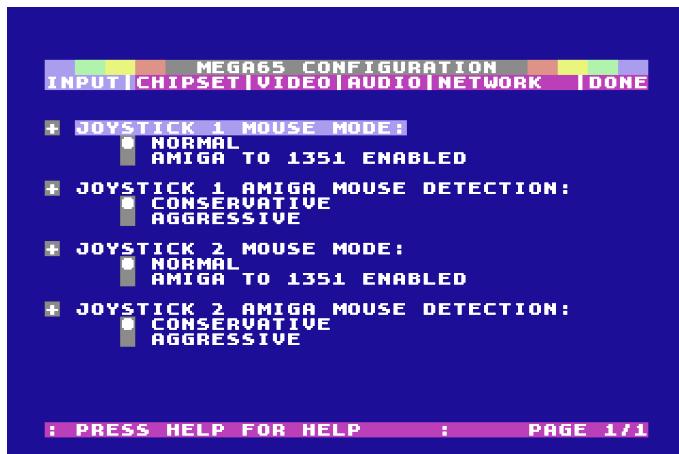
CONFIGURATION UTILITY

The configuration utility for the MEGA65 has a similar purpose to the BIOS on a PC, which is to allow you to control certain default behaviours of your MEGA65. However, rather than storing the configuration data in battery-backed RAM, the MEGA65 stores this data on sector 1 of the SD card. If you switch SD cards, you will change the configuration data.

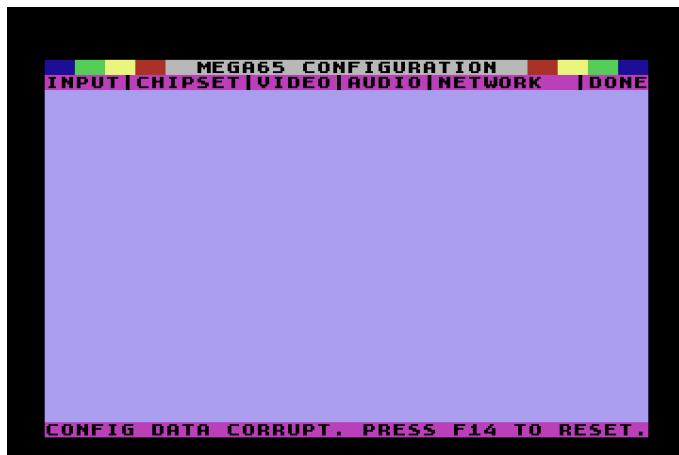
To enter the configuration utility, switch the MEGA65 on while holding **ALT** down. This will show the utility menu, similar to the following:



Next, press the number corresponding to the **CONFIGURE MEGA65** item. The MEGA65 Configuration Utility will launch, showing a display similar to the following:



If your MEGA65's System Partition is corrupted, you may be prompted to press **F14** to correct this, (i.e., hold **SHIFT** and press **F13**), with a display similar to the following:



To correct this error, press **F14**. Next, press **F7** to save the reset configuration, otherwise the reset data will not be saved to the MEGA65 System Partition.

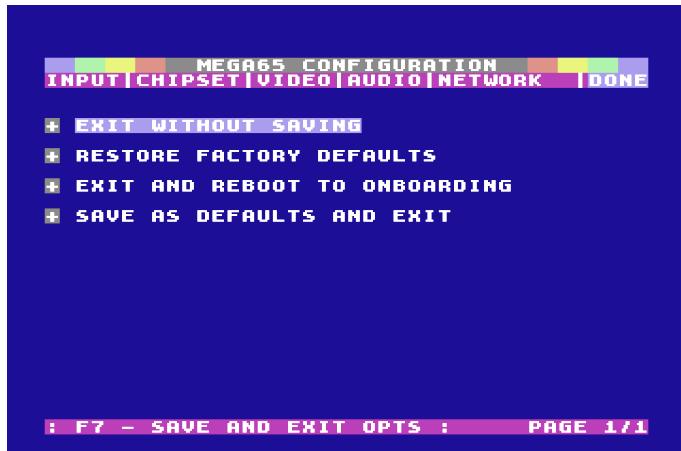
Once you have dismissed that display, or if your MEGA65 System Partition is not corrupt, you can begin exploring and adjusting various settings. The program can be controlled using the keyboard, or optionally, a 1351 or Amiga™ mouse.

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

Use **↑** and **↓** to select an item.

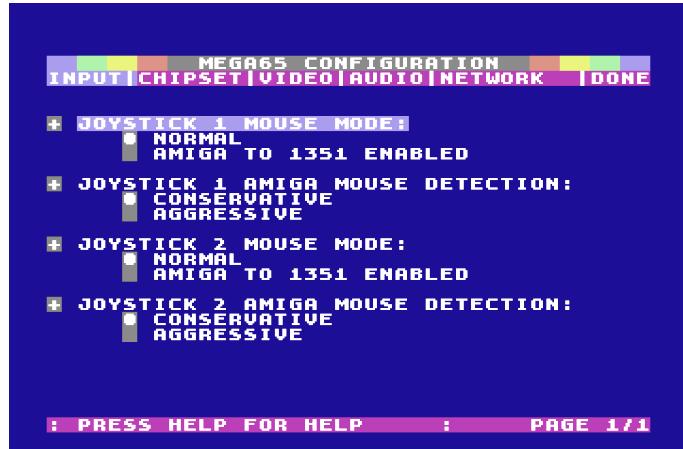
Press **RETURN** or **SPACE** to toggle a setting, or to change a text or numeric value. The black circle next to an option indicates the current selection.

When you have finished, press **F7** to see the options for saving the changes. This will give you four options:



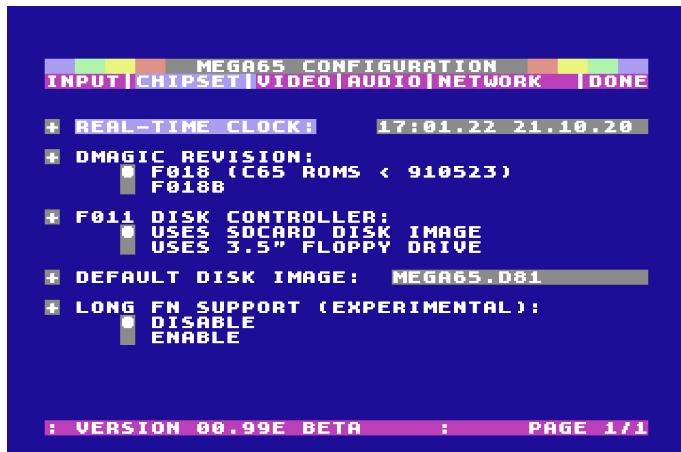
- **EXIT WITHOUT SAVING** abandons any changes made in the MEGA65 Configure utility and exits.
- **RESTORE FACTORY DEFAULTS** resets the MEGA65 configuration settings to the factory defaults. It will randomly select a new MAC address for models that include an internal Ethernet adaptor. If you wish to commit these changes, you must still save them.
- **EXIT AND REBOOT TO ONBOARDING** abandons any changes made in the MEGA65 Configure utility and reboots to the Onboarding utility.
- **SAVE AS DEFAULTS AND EXIT** commits changes made to the SD card. These changes will be used when the MEGA65 is switched on.

Input Devices



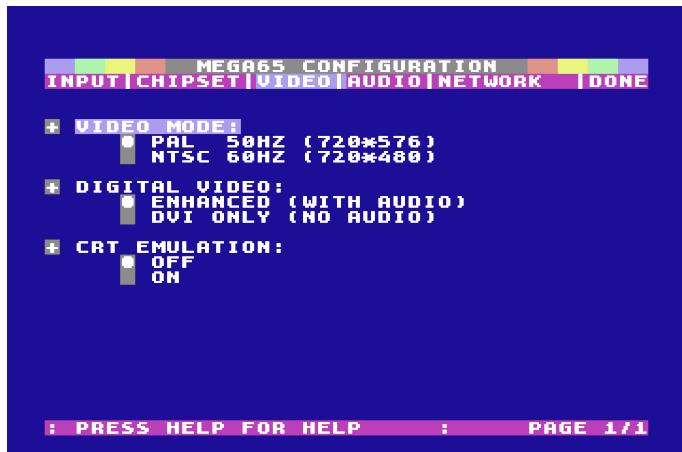
- JOYSTICK 1 AMIGA MOUSE MODE allows either **NORMAL** operation, where software will see it as an Amiga mouse or **1351 EMULATION** mode, where the MEGA65 translates the Amiga mouse's movements into 1351 compatible signals. This allows you to use an Amiga mouse with existing C64/C65 software that expects 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 misbehaving, you might want to use the **CONSERVATIVE** option.
- JOYSTICK 2 AMIGA MOUSE MODE is identical to the first option, but for the second controller port. This allows you to have different settings for each port.
- JOYSTICK 2 AMIGA MOUSE DETECTION similarly provides the ability to separately control the Amiga mouse detection algorithm for the second controller port.

Chipset



- **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 **RETURN**. The display does not change while viewing this page, but if you use **←** and **→** to select another page and return to this page, the values will update if a Real-Time Clock is fitted and functioning.
- **DMAgic REVISION** allows selecting the default mode of operation for the C65 DMAgic DMA controller. This option is only required for ROMs not detected by the MEGA65's HYPPPO 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 **USES SDCARD DISK IMAGE**. You can read more about D81 disk images on page [6-6](#).
- **LONG FN SUPPORT** is a feature that is still under development at time of writing, and we suggest leaving it disabled for now until the feature matures in future bitstreams. Its aim is to provide long filename support for the SD card.

Video



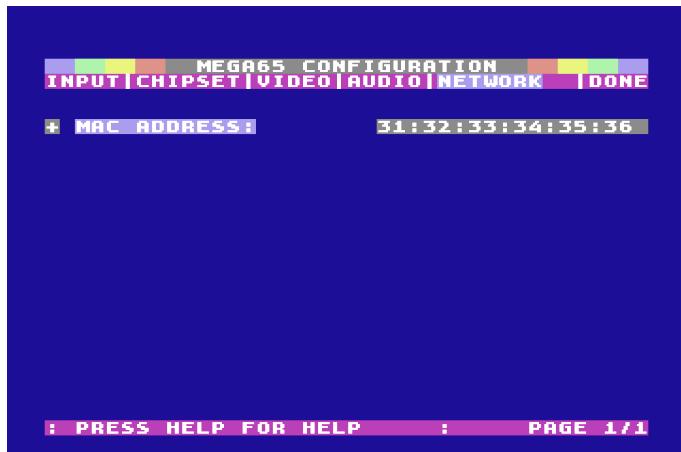
- **VIDEO MODE** selects whether the MEGA65 starts in PAL or NTSC. The MEGA65 supports true 480p NTSC and 576p PAL double-scan modes, with exact 60Hz / 50Hz frame-rates. This setting sets the default value, and the system can be switched between PAL and NTSC via the Freeze Menu, or under software control by MEGA65-enabled programs.
- **DIGITAL VIDEO** allows for selection between either **ENHANCED** video output containing audio, or **DVI ONLY** video output with no audio.
- **CRT EMULATION** selects whether CRT scanline emulation should be applied to the video output or not.

Audio



- **AUDIO OUTPUT** selects whether the SIDs and digital audio channels are combined to provide a monaural signal or whether the left and right tagged audio sources are separated to provide a stereo signal. This setting can be changed 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-hand sides of the stereo audio output. This is useful for software that expects left and right SIDs to be at swapped addresses compared with the MEGA65 defaults.
- **DAC ALGORITHM** allows selecting between two different digital to analog conversion algorithms. Both options sound good and the selection is a personal preference.
- **AUDIO AMPLIFIER** allows enabling or disabling the audio amplifier contained in some models of the MEGA65. This option works for 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.

CHAPTER 5

Cores and Flashing

- **What are Cores, and Why Do They Matter?**
- **Bitstream files**
- **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**

- **DIP Switches**

WHAT ARE CORES, AND WHY DO THEY MATTER?

The MEGA65 computer uses a versatile chip called an FPGA as its heart, which is an acronym for "Field Programmable Gate Array". This is a fancy way of saying that FPGAs are chips that can be programmed *by you* to impersonate other chips. They do this by re-configuring their arrays of logic gates to reproduce the circuits of other chips. As a result, FPGAs are not an emulation, but a re-creation of other chips.

However, FPGAs forget what chip they are pretending to be whenever the power is turned off, or when they are re-programmed. This might sound annoying, but it's actually very powerful. It means that you can tell the FPGA in the MEGA65 to impersonate not just the MEGA65 design as it currently stands, but to impersonate any improvements made to the design itself. In other words, you can upgrade the MEGA65 hardware just by providing a new set of instructions to the FPGA. These sets of instructions are called "cores", or "bitstreams". For the purpose of the MEGA65, these two terms are 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 to be other interesting home computers. We believe that the FPGA is powerful enough to re-create a Commodore PET™, VIC-20™, Apple II™, Spectrum™, BBC Micro™, or even an Amiga™, 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 various 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 system, and back again.

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

Model types

Retail models of the MEGA65 are referred to as the MEGA65R3A (revision 3A). Throughout the course of development of the MEGA65, there have been several other model variants used by developers, each with differing specifications and available core slots, so they will be listed here, just to raise awareness of them.

Model	FPGA type	QSPI size	#slots	slot size
MEGA65R3A	The retail/release version of the MEGA65			
	A200T	64MB	8	8MB
MEGA65R3	The DevKit model			
	A200T	32MB	4	8MB
MEGA65R2	An earlier MEGA65 model			
	A100T	32MB	8	4MB
Nexys4	FPGA development boards used early in the project			
	A100T	16MB	4	4MB

BITSTREAM FILES

Firstly, there are a variety of files related to the MEGA65's cores/bitstreams that you should be familiar with, in order to decide what file-types are needed for what occasion.

File types

File-type	Purpose
.cor	The MEGA65 project's custom bitstream file format, containing extra header information to help identify the bitstream and the specific MEGA65 target device it is intended for. The MEGA65's flashing utility makes use of this additional information to ensure you don't accidentally flash the bitstream of a different device.
.mcs	The bitstream file in a format needed when flashing it to your device's QSPI flash memory chip via Vivado®.
.prm	This file contains checksum information that can be used by Vivado to verify the .mcs file you have tried to flash. Optional.
.bit	A plain bitstream file in its purest binary form.

Where to Download

Visit the following url:

<https://files.mega65.org>

The screenshot shows the FILEhost website interface. At the top, there are three tabs: 'News', 'Files' (which is highlighted with a pink oval and has a pink arrow pointing to it), and 'Articles'. Below the tabs is a search bar with the text '.cor' typed into it, and a magnifying glass icon. The main content area is a table listing files. The columns are 'Title', 'Download', 'Rating', 'Category', and 'Type'. The rows contain the following data:

Title	Download	Rating	Category	Type	
nexys4-dev.cor		16	-	Firmware	COR File
nexys4ddr-widget-dev.cor		22	-	Firmware	COR File
mega65r3-dev.cor		81	-	Firmware	COR File
mega65r2-dev.cor		12	-	Firmware	COR File

Click the **Files** tab, and in the search-bar, type **.cor** and press Enter. For the purposes of this chapter on core-flashing, download the desired .cor file that suits your target device:

- **mega65r3-dev.cor** (for MEGA65R3 boards, both Release and DevKits)
- **mega65r2-dev.cor** (for MEGA65R2 boards)
- **nexys4ddr-widget-dev.cor** (for Nexys4 DDR boards)
- **nexys4-dev.cor** (for Nexys4 PSRAM boards)
- You can also find .bit, .mcs and .prm files located here too.

Alternatively, if you intend to flash the QSPI chip via Vivado, you would instead download the .mcs file for your target device (and optionally, the .prm files as well).

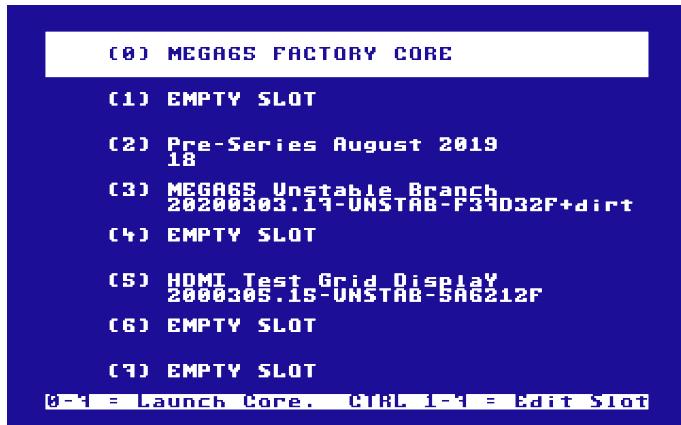
Another alternative for Nexys4 board users is to download .bit files and copy them to SD cards, which you can also download.

But once again, for the purposes of this chapter on core-flashing, you will only be interested in the .cor files.

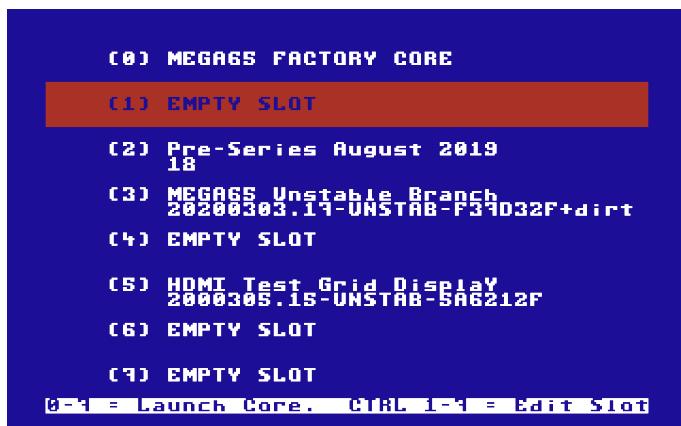
SELECTING A CORE

To operate the MEGA65 with an alternate core, switch off the power to the MEGA65, and then hold **NO SCROLL** down while switching the power back on. This instructs the

MEGA65 to enter the Flash and Core Menu, instead of booting normally. When booting this way, the following screen will appear:



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



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 back into the menu automatically.

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

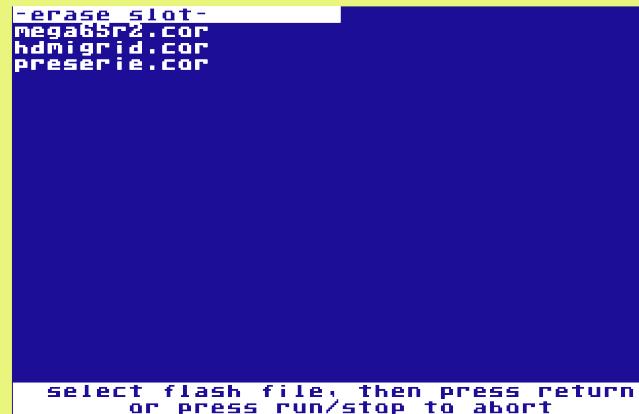
Installing and upgrading the core (from a .cor file) for the MEGA65 can be done in a few easy steps.

First, copy the core into the MEGA65 SD card's root directory. You can do this by removing the SD card and copying a previously downloaded core file to it from another 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 upload the core upgrade file onto the SD card from another computer on your local network.

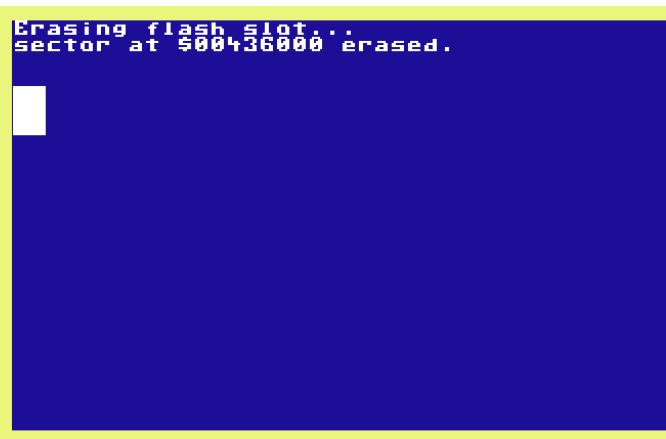
The Flash Menu will use the external microSD slot over the internal SD card, so if you have both a microSD card and SD card inserted in your MEGA65, the Flash Menu will ignore the internal SD card. To avoid this, simply copy the core(s) from the internal SD card to the external microSD card, or temporarily 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 only supports DOS-style 8.3 character filenames in UPPERCASE. If your core files have a longer name, you will need to rename them when copying them onto your microSD or SD card.

Note: If you plan on using your external microSD card solely for upgrading cores (and not for booting from), then you will not need to add "SD card essentials" files onto your microSD card (simply adding the .cor file will suffice).

Next, once you have the upgrade core on the MEGA65's SD card, enter the Flash and Core Menu as above, i.e., switch off the power, and hold **NO SCROLL** down while switching the power on again. When the Flash and Core Menu appears, hold **CTRL** down and press **1** (or **CTRL** and a different number, if you wish to replace the contents of a different flash slot). The MEGA65 will present you with a list of core files that are on the SD card:



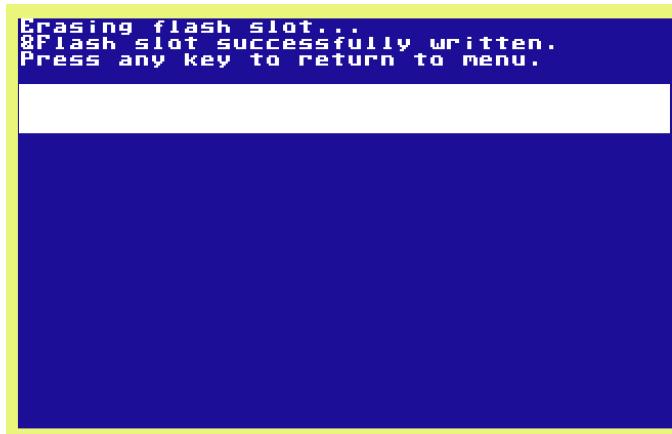
Select the upgrade core file you wish to install using the cursor keys, and then press **RETURN**. The MEGA65 will then erase the flash slot, before writing the upgraded core. You will see a progress bar while the MEGA65 erases the flash slot:



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 then perform only the erasure part of the process.

It is important to not switch 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, it won't damage your MEGA65 or leave it in an unusable state: It will simply 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 has completed, you will see a message indicating that the process is complete:



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

Note: After flashing completes, it is safe to delete .cor files from your SD card, as their contents now reside within the QSPI flash memory.

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 to another slot.

Of course, there is nothing stopping you from installing a different core in slot 1, so that the MEGA65 behaves as a different type of computer when you switch 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 Chapter/Appendix T

on page T-5, 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 it ever gets corrupted, it will “brick” the machine. This will require you to connect a TE-0790 JTAG programmer, by opening your MEGA65 case, installing the module, going through some rather convoluted software preparation steps (similar to if you were creating your own bitstream/-core) and then restoring a working bitstream into the slot.

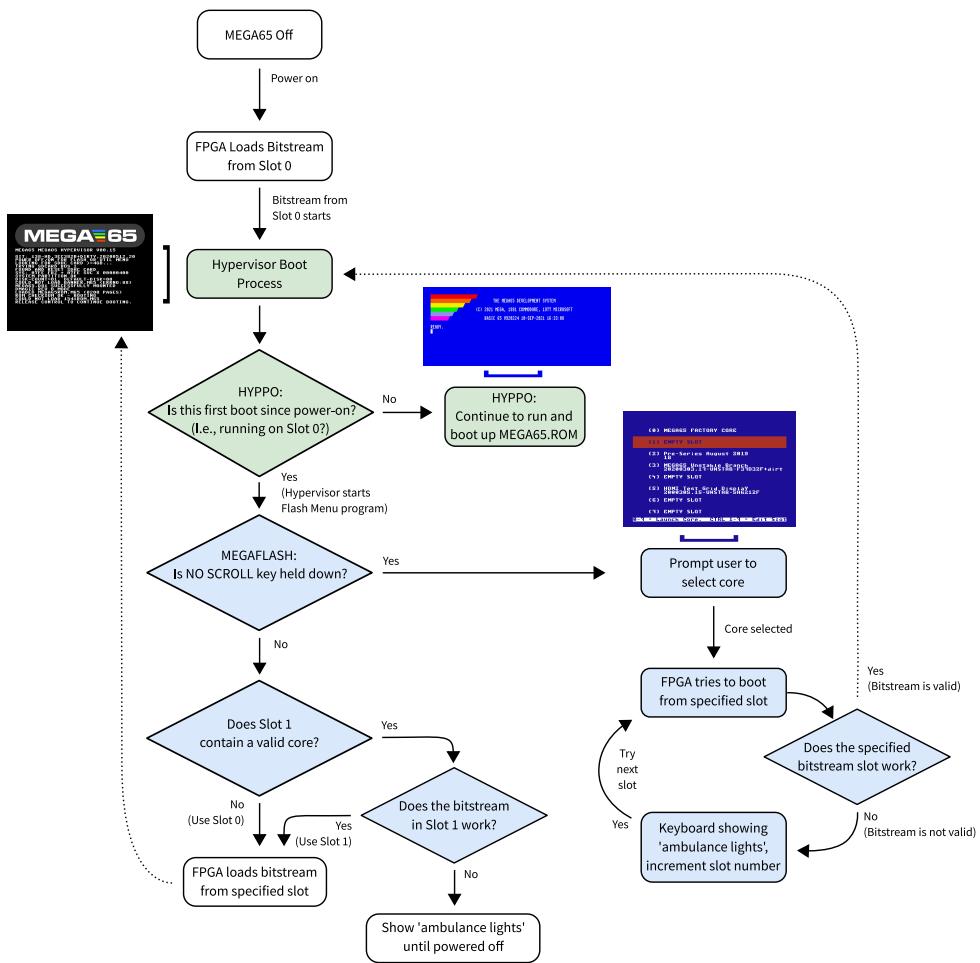
The MEGA65 is an open system though, so it’s possible for you to do all of this, but it’s very hard. There is a secret key-press combination in the Flash Menu that will then challenge you with a series of questions with increasing difficulty to ensure that you know what you are doing. Only after you have correctly answered these questions will you be given the option to erase and/or replace the contents of slot 0. Details of the questions asked are purposely not documented.

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 write them to other slots and use the Flash Menu to activate them, or you can simply use a TE-0790 JTAG programmer, and then use Vivado or other FPGA development tool to write to the flash directly. This 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 with when it is switched on. The process is shown in the following figure:



The booting process is governed by two facilities:

- The Hypervisor (also known as HYppo), which operates at a level above the Kernel. One of its responsibilities is to manage aspects of the boot process. For more details on the Hypervisor, refer to Chapter/Appendix G on page G-24. In the diagram, activities performed by the Hypervisor have been highlighted in green.
- The Flash Menu program (also known as MegaFlash), which provides a list of available core slots to choose from. In the diagram, activities performed by MegaFlash have been highlighted in blue.

When the MEGA65 is switched on, it does the following:

- Loads the bitstream stored in slot 0 of flash memory. If that is the MEGA65 Factory Core, the MEGA65 HYPOPO Hypervisor starts.
- If it is the first boot since power-on (which implies that we are running from slot 0), HYPOPO starts the Flash Menu program (aka MegaFlash) – 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 then checks if  is being held down.
- If it is, the Flash Menu program shows its display, allowing you to select or re-flash a core.
- If  is not being held down, the Flash Menu program checks if Flash Slot 1 contains a valid core.
- If it does, then the Flash Menu program attempts to load that core.
- If it succeeds, then the system reconfigures itself for that core, after which the behaviour of the system is according to that core.
- If it fails, the keyboard will go into “ambulance mode”, showing flashing blue lights to indicate that some first-aid is required. Note that in ambulance mode the reset button has no effect: You must switch the MEGA65 off and on again.

If you have 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.

DIP SWITCHES

The MEGA65 motherboard has 4 DIP switches (at position S3 on the PCB), which are documented in the following table.

Switch#	Purpose	Default
1	Off = Disable Keyboard Flash On = Enable Keyboard Flash	Off
2	Off = Disable remote control via Ethernet	Off = Enable remote control via Ethernet
3	Off = Disallow non-hypervisor access to QSPI flash On = Permit non-hypervisor access to QSPI flash	Off
4	Off = Boot from slot 1 On = Boot from slot 2	Off

6

CHAPTER

Floppy Disks, the Freezer, and D81 Images

- Disk Drives
- The Freezer
- Using D81 images

DISK DRIVES

The MEGA65 is compatible with a wide range of floppy disk drives, and can be configured to use them in a variety of different ways. It includes an internal 3.5" drive, which is compatible with double density and high density 3.5" disks. But did you know you can also use 5.25" disks? By connecting a compatible IEC drive to the rear Disk Drive/IEC Connector, you can use your MEGA65 with two or more physical floppy drives simultaneously!

There are also IEC drive emulators available, such as the SD2IEC. These devices (and the MEGA65) use "disk images", which are files that represent a floppy disk, usually stored on an SD card. By "mounting" a disk image file to a disk drive emulator, the MEGA65 can then use these files without realising it's actually being emulated. As far as the MEGA65 is concerned, it's using a real drive.

This allows you to use modern media to read and write your programs, and in some cases can drastically reduce read and write times. These files usually have the extension **.d64** (named after the C64), which are files that represent a *single* side of a 5.25" disk, or **.d81**, which represents an entire 3.5" disk (named after the 1581 floppy disk drive).

The MEGA65 can use either real disks and drives, **.d81** files, or a combination of both.

Disk Drive Terminology

Before you get to know how to use the various disk drives that work with the MEGA65, it's important to be aware of some specific terminology which CBM computers and the MEGA65 use. This includes **BASIC** commands (such as **LOAD** and **SAVE**), and **CBDOS** (Computer Based Disk Operating System), which use the following:

UNIT is a device number in the range of 0-31. The numbers from 0 to 11 are reserved for the following device types:

Unit #	Device	Notes
0	Keyboard	Input
1	Unused	Was used for Cassettes on the C64
2	Unused	Was used for RS-232 on the C64
3	Screen	Input/Output
4-5	IEC Printer	Output
6-7	IEC Plotter	Output
8-9	CBDOS drives ¹	Internal floppy drive, 1565 ² , or disk image
10-11	IEC drives	1541, 1571, 1581, FD-2000

DRIVE is the drive number of a **UNIT**:

Unit	Drive Numbers	Comment
1541 IEC	0	Single drive
1571 IEC	0	Single drive
1581 IEC	0	Single drive
FD-2000 IEC	0	Single drive (CMD)
FD-4000 IEC	0	Single drive (CMD)
SD2IEC	0	Disk images
4040 IEEE-488	0,1	Dual drive (CMD)
8050 IEEE-488	0,1	Dual drive (CMD)
8250 IEEE-488	0,1	Dual drive (CMD)

For all single drives, the drive number is always 0. This includes all of the well known drives with an IEC interface.

Dual disk drives (which use drive numbers 0 and 1) are usually equipped with an IEEE-488 interface, and need an IEEE-488 to IEC converter to be used on the MEGA65.

The internal floppy controller of the MEGA65 can be used to control two floppy drives. However, both drives must be attached to the same ribbon cable.

BASIC commands that address files or disks include **U** for UNIT and **D** for drive, which use the default values of **UNIT = 8**, and **DRIVE = 0**.

THE FREEZER

The MEGA65 **FREEZER** is a tool for changing system parameters at any time, regardless of the currently running program. It's very similar to the Freezer that many popular cartridges for the C64 included, such as the Action Replay cartridge, and The Final cartridge. Freezers generally work by taking over the CPU whilst leaving RAM intact. This way you can perform functions and change configuration settings without affecting the currently running program. Once you have finished using the Freezer, CPU control is given back to the original program.

The Freezer is invoked by pressing and holding  for half to one second. The current status of the computer is frozen and the Freezer menu, similar to the following picture is displayed. This chapter focuses on how to assign disk images and configure the internal floppy disk drive, but the other options in the Freezer are self explanatory, or will be covered in detail in online documentation.

¹IEC drives can be assigned to devices 8-9, by moving the built-in MEGA65 devices to 10-11 first. Once the drives have been reassigned, the IEC drives can be switched on.

²The 1565 was a prototype external C65 disk drive that never made it to production.



The bottom-right part of the Freezer screen shows the current disk drive assignments. The internal CBDOS which the MEGA65 uses supports up to two 3.5" disk drives and .d81 images, in any combination. Drive 0 can be assigned to the internal floppy disk drive or a .d81 disk image, whereas Drive 1 can be assigned to an external floppy disk drive (connected to the same ribbon cable as the internal drive), or a .d81 disk image.

A typical configuration is usually one of the following:

Drive 0 Assignment	Drive 1 Assignment
Internal floppy disk drive	Disk image
Disk image	Disk image

The assignment, and mounting of disk images can be performed by pressing **0** for drive 0, or **1** for drive 1.

You may have noticed that the drive numbers here (0-1) are different to the more commonly used **UNIT** numbers (8-9), which you may be more used to. The drive numbers of 0 and 1 are used internally by CBDOS. However, **BASIC** and Kernal address these storage devices by **UNIT** numbers. CBDOS emulates two single drives for the operating system, by assigning separate unit numbers to drive 0 and drive 1. The default assignment is:

Assignment	Unit/Drive
Internal drive 0 (internal floppy or disk image)	UNIT 8, DRIVE 0
Internal drive 1 (external floppy or disk image)	UNIT 9, DRIVE 0

You may wish to change this unit assignment, if for example a 1541 drive is connected to the IEC port as unit 8. In this case, the internal drive assignment can be switched to an alternative unit to avoid conflict. To do this, press **8** to toggle the unit assignment

of drive 0 between 8 and 10, and **9** to toggle the unit assignment of drive 1, between 9 and 11.

After setting the preferences, the Freezer can be exited with **F3**. The Freezer restores the screen and returns to the interrupted program.

Note: The drive and unit assignments are temporary, and will be reset to their defaults after power down or reset. For permanent settings, use the **CONFIGURE** menu. You can read more about this menu on page [4-16](#).

USING D81 IMAGES

As previously mentioned, the MEGA65 allows the use of D81 files instead of using physical disks. The MEGA65 includes a built-in file browser, which allows you to choose which D81 file to mount. To do this, enter the Freezer menu, then press **0** or **1**, to choose the drive to mount it to.

When you do this, the file browser will appear:



The file browser will list all D81 images on the microSD card. The MEGA65 prioritises the external microSD card, so if you prefer to use an image on the internal SD card, the microSD card should be removed (when doing this, the MEGA65 needs to be reset as well).

Press the cursor keys to choose a file. If the currently selected file isn't a valid D81, the border will be red, otherwise it will be blue, as shown above. Whilst browsing for D81 files, the list of files within the currently selected D81 file will be shown on the right-hand side.

You may have noticed that the file browser has a - **NO DISK** - , - **NEW D81 IMAGE** - and a - **INTERNAL 3.5"** - entry. - **NO DISK** - is used to tell the MEGA65 that no disk is inserted, and to not try and find one when performing a disk-based operation.

- **NEW D81 IMAGE** - creates a new, blank D81 image to use. When you select this option, you'll be asked to enter the filename for the new image, which is limited to 8 characters in length.

- **INTERNAL 3.5"** - is used when using a physical 3.5" disk in the internal drive. If no disk is inserted and a disk-based operation is performed with the - **INTERNAL 3.5"** - option, the MEGA65 will try and find a disk, which could take a few seconds. Any previously mounted D81 files will be unmounted when using these options.

When you find the D81 file you wish to mount, press **RETURN**. This will take you back to the Freezer menu, and the selected D81 file will be shown under the selected drive. In the following screenshot, the **MEGA65.D81** file has been mounted to drive 0:



Next, press **F3** to exit the Freezer menu and return to the currently running program. Alternatively, pressing **F5** will exit the Freezer, and reset the MEGA65 as well.

Now that the file has been mounted, you can use the **DIR** command to list the files in the image, or any of the **LOAD** and **SAVE** commands, like you would on a physical disk.

Auto Booting From a Disk

If you would like the MEGA65 to automatically perform any BASIC commands upon startup, you can create an **AUTOBOOT.C65** PRG file on your disk/image. In this file,

you can write BASIC programs to do things such as changing the **BACKGROUND**, **BORDER**, or **foreground** colours, or even **LOAD** a program.

For example, to change the background and border colours to black, and load a program called "MYPROGRAM" automatically, create a BASIC listing like the following, then **SAVE** it:

```
10 BACKGROUND 0  
20 BORDER 0  
30 LOAD "MYPROGRAM.PRG"  
SAVE "AUTOBOOT.C65"
```

Saving D81 file settings

The D81 settings are temporary, and will be reset to their defaults after power down. You can change the default settings in the **CONFIGURE** menu. When opening the Configure menu, press the **F1** and **F2** keys to reach the **CHIPSET** section, then navigate to the **F011 DISK CONTROLLER** options. Select the **USES SD CARD DISK IMAGE** setting, then enter the name of the disk image you wish to have mounted by default in the **DEFAULT DISK IMAGE** field. A screenshot showing these settings with **MEGA65.D81** as the default D81 disk image is shown:



You can read more about the Configure menu on page [4-12](#).

PART III

FIRST STEPS IN CODING

CHAPTER

7

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 programs 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 programs 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 program, 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 Program Counter, or PC, for short (not to be confused with PC when people are talking about a Personal Computer). The PC is just the number of the next of the next memory location (or in our case, egg-cup), that the computer will examine, when deciding what to do next. You might like to have another piece of paper that you can use to write the PC number on as you go along.
- Third, we need to have a list of things that the egg-cup computer will do, based on what number is in the egg-cup indicated by the PC.

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

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

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

CHAPTER

8

Getting Started in BASIC

- Your first BASIC programs
- 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/Appendix 3 on page 3-3, 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 switch 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 program the MEGA65 as we have had making it!

YOUR FIRST BASIC PROGRAMS

The MEGA65 was designed to be programmed! When you switch 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 programs - 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 uppercase mode, so you don't need to hold **SHIFT** or **CAPS LOCK** down. When you have typed "HELLO COMPUTER", press **RETURN**. 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 program. 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 **SHIFT**, and then pressing **2**. Don't forget to press **RETURN** 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 **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/Appendix [3 on page 3-3](#). But for now, if you make a mistake just press **RETURN** and type the messed up line again.

Hopefully now you will see something like the following:

The screenshot shows the Commodore C65 Development System interface. At the top, it displays "THE COMMODORE C65 DEVELOPMENT SYSTEM", "COPYRIGHT 1991 COMMODORE ELECTRONICS, LTD.", and "COPYRIGHT 1977 MICROSOFT". Below that, it says "BASIC 10.0 V0.9B.911001 ALL RIGHTS RESERVED". The main area of the screen shows the following text:
ENGLISH KEYBOARD
NO EXPANSION RAM
C1565 DRIVE
ROM CHECKSUM \$4BCF
READY,
HELLO COMPUTER
?SYNTAX ERROR
READY
PRINT"HELLO COMPUTER"
HELLO COMPUTER
READY.

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:

```
COPYRIGHT 1991 COMMODORE ELECTRONICS, LTD.  
COPYRIGHT 1977 MICROSOFT  
BASIC 10.0 V0.9B.911001 ALL RIGHTS RESERVED  
ENGLISH KEYBOARD  
NO EXPANSION RAM  
C1565 DRIVE  
ROM CHECKSUM $4BCF  
READY.  
HELLO COMPUTER  
?SYNTAX ERROR  
READY.  
PRINT "HELLO COMPUTER"  
HELLO COMPUTER  
READY.  
PRINT HELLO COMPUTER  
8  
READY.
```

In the above examples we typed commands in directly, and the computer executed them immediately after you pressed **RETURN**. 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 program. 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, at the left-hand 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 program, a bit like the program 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 program.

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 program, we now know that the 2nd command will try to print the piece of information stored in the variable **I**. So let's 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 3rd 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 let's 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 2nd 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 program of commands we typed in. We do this by using the **RUN** command. Because we want it to run the program immediately, we should use immediate mode (remember, this is another name for direct mode). So just type in the word **RUN** and press **RETURN**. You should then see a display that looks something like the following:

BASIC 10.0 V0.9B.911001 ALL RIGHTS RESERVED

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

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

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 program. 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 program is quite safe for now. We can see that this is the case by typing the **RUN** command a couple more times: The program listing will have scrolled off the top of the screen, but we can still **RUN** the program, because the computer has remembered it. Give it a try! Did it work?

If you wish to see the program of remembered commands, you can use the **LIST** command. This command causes the computer to display the remembered program of commands to the screen, like in the display here. If you would like to replace any of the commands in the program, 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 program to **PRINT I;** by typing the following line of input and pressing **RETURN**:

```
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 program, like this:

The screenshot shows a sequence of interactions with a BASIC interpreter:

- The first screen shows the numbers 7, 8, 9, and 10 listed vertically.
- The second screen shows the command **READY.** followed by the command **LIST**. The output displays the lines of code:

```
1 FOR I = 1 TO 10 STEP 1
2 PRINT I
3 NEXT I
```
- The third screen shows the command **READY.** followed by the command **LIST**. The output displays the lines of code:

```
1 FOR I = 1 TO 10 STEP 1
2 PRINT I;
3 NEXT I
```
- The fourth screen shows the command **READY.** followed by the command **RUN**. The output displays the numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

It is quite easy to modify your programs 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 program. Alternatively, you can give a range separated by a minus sign to display only a section of the program, e.g., **LIST 1 - 2** to list the first two lines of our program.

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 **RETURN** 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 program at the same time, you must press **RETURN** on each line that has been modified. It is good practice to check that the program 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 1×1 to 10×10 ?

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 it refers to each symbol 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 things 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 **M**, and pressing **RETURN**, and see what happens. And then let's try with a string, say "**THIS IS A STRING**".

THE COMMODORE C64 DEVELOPMENT SYSTEM
COPYRIGHT 1991 COMMODORE ELECTRONICS, LTD.
COPYRIGHT 1977 MICROSOFT
BASIC 10.0 V0.9B.911001 ALL RIGHTS RESERVED

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

READY.
M

?SYNTAX ERROR
READY
"THIS IS A STRING"

?SYNTAX ERROR
READY.
```

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 stored in a variable using the **PRINT** command. So instead, we could type in **PRINT M**, and the computer would know what to do, and will print the piece of information stored in the variable called **M**.

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 shortcut: If you type a question mark character, i.e., a ?, the computer knows that you mean **PRINT**. So for example if you type ? **M**, it will do the same as typing **PRINT M**. Of course, you have to press **RETURN** after each command to tell the computer you want it to process what you typed. From here on, we will assume that you 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 program. So if you type **I ? N**, and then **LIST**, you will see **I PRINT N**, as we can see in the following screenshot:

```
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 wanting 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 program 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 program. 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:

The screenshot shows the Commodore C64 Development System BASIC 10.0 interface. The screen displays the following text:

```
THE COMMODORE C64 DEVELOPMENT SYSTEM
COPYRIGHT 1991 COMMODORE ELECTRONICS, LTD.
COPYRIGHT 1977 MICROSOFT
BASIC 10.0 V0.9B.911001 ALL RIGHTS RESERVED

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

READY.
INPUT A$

?ILLEGAL DIRECT ERROR
READY.
1 INPUT A$
2 PRINT "YOU TYPED "; A$
RUN
? ■
```

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 program 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 program 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 **RETURN** to tell the computer that you are done, the program will continue, like this:

The screenshot shows the Commodore C65 Development System interface. At the top, it displays "THE COMMODORE C65 DEVELOPMENT SYSTEM", "COPYRIGHT 1991 COMMODORE ELECTRONICS, LTD.", and "COPYRIGHT 1977 MICROSOFT". Below that, it says "BASIC 10.0 V0.9B.911001 ALL RIGHTS RESERVED". The screen then shows system information: "ENGLISH KEYBOARD", "NO EXPANSION RAM", "C1565 DRIVE", and "ROM CHECKSUM \$4BCF". It then displays a series of commands and their results:
READY.
INPUT A\$
?ILLEGAL DIRECT ERROR
READY.
1 INPUT A\$
2 PRINT "YOU TYPED "; A\$
RUN
? LIGHT SABRE
YOU TYPED LIGHT SABRE
READY.

Of course, we didn't really know what to type in, because the program 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 program, 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 program, 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 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 program, 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 program. It exists only in the variable. This ability to process information that is not part of a program is one of the things that makes computer programs 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 program we run it again and again, and give it different input each time, and the program 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
? KRYTON
HELLO KRYTON

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

READY.
```

We can see that each time the program 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 program 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 program 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 program 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 program, 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 program:

```
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 program 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 program, 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 program:

```
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.  
RUN  
WHAT IS YOUR NAME AND AGE  
? 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 program 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 program 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 **RETURN** 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 program. For example:

```
LIST
1 PRINT "WHAT IS YOUR NAME AND AGE"
2 INPUT A$,A,B
3 PRINT "HELLO "; A$
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:

```
1
```

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$,B
LIST
1 PRINT "WHAT IS YOUR NAME AND AGE"
2 INPUT "WHAT IS YOUR NAME AND AGE",A$,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 program 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 programs don't get all cluttered up with empty lines if you make lots of changes to your programs.

It is also possible to **DELETE** a range of lines. For example (but don't do this now), you could delete lines 3-4 with:

DELETE 3-4

You can read more about the **DELETE** command in the BASIC 65 Command Reference.

With that out the way, let's run our program and see what happens. As usual, just type in the **RUN** command and press **RETURN**. 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.
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 program, 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 program 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 program 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 program 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 program expects both pieces of information at the same time. This means the program can't print a message about the first message until after it has both pieces of information. Change the program 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 program, so that you can have more than one **INPUT** and **PRINT** command.

3. Can you write a program 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 program that asks them, and remembers the answers and prints them out with an appropriate message. For example, running your program 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 program, 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 programs 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 <> 3** would be true, while **3 <> 3** would be false.
- Less than or equal to (\leq). For example, **1 < 3** and **3 <= 3** would be true, while both **4 < 3** would be false.
- Greater than or equal to (\geq). For example, **3 >= 1** and **3 >= 3** would be true, while both **1 >= 3** would be false.

A good trick if you have trouble remembering which way the ($<$) and ($>$) 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-hand side needs to be less than the number on the right-hand 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 ($>=$), it is true if the condition is true for any of the symbols in it. So in this case the ($>$) 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:

```
COPYRIGHT 1977 MICROSOFT
BASIC 10.0 V0.9B.911001 ALL RIGHTS RESERVED

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

READY.
IF -2 < 0 THEN PRINT "-2 IS A NEGATIVE NUMBER"
-2 IS A NEGATIVE NUMBER

READY.
IF 2 < 0 THEN PRINT "2 IS A NEGATIVE NUMBER"

READY.
IF 0 < -2 THEN PRINT "-2 IS A POSITIVE NUMBER"

READY.
IF 0 < 2 THEN PRINT "2 IS A POSITIVE NUMBER"
2 IS A POSITIVE NUMBER

READY.
```

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 program. So let's make a little program 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 program, have a think about how you might do it, and see if you can figure it out. The clue we 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 program 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 we 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 program 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:

```
I 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 program, 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, the programmer can create very flexible programs.

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 program yourself.

Here is how we 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 program 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 **SHIFT**, and then press **CLR HOME**, it clears the screen. This is much simpler and more convenient. But how can we do something like that in our program? 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. We 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 **ll** symbol appears. If you press the cursor right key, a **u**, up **o**, down **u** and the **CLR HOME** a **g**, and if you are holding down **SHIFT** and press **CLR HOME** a **u**.

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 press **CLR HOME**. 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 program. 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 program by typing in **RUN** and pressing  as usual, the 2nd 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, the same as 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 program again.

There are a few ways we could do this. We already saw the **FOR - NEXT** pattern. With that, we could make the program give the user a certain number of guesses. If we followed the **NEXT** command with another program line, we could even tell the user when they have taken too many guesses. So let's have a look at our program 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 program, 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 program, 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 let's 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 "U 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 "U 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 program will keep going. We'd really like the program 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 program, the computer tells you where it has stopped, and you have the chance to continue the program using the **CONT** command. The **END** command, on the other hand, tells the computer that the program has reached its end, and it should go back to being **READY**. The **END** command makes more sense for our program, because after the user has guessed the number, there isn't any reason to continue.

Now we need a way to 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 program, 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 program, 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 program. 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 program:

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

If you type those lines in, and **LIST** the program 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 program, and see whether it worked. Let's try it!



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

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 program, we can now make guesses, and the program 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 program 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 program ask at the start for the secret number?

At the moment the program 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 program ask for the user's name and give personalised responses?

At the moment, the program 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 program 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 program 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 program say if a guess is “warmer” or “colder” than the previous guess?

At the moment the program 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 program 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  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 the MEGA65 BASIC 10.0 terminal window. At the top, it displays "BASIC 10.0 V0.98.911001 ALL RIGHTS RESERVED". Below that, system information is shown: "ENGLISH KEYBOARD", "NO EXPANSION RAM", "C1565 DRIVE", and "ROM CHECKSUM \$4BCF". The window then shows four separate runs of the command `PRINT RND(1)`, each resulting in a different floating-point number between 0 and 1. The first run shows scientific notation: `1.07870447E-03`. The subsequent runs show `.793262171`, `.44889513`, and `.697215893`. Finally, the terminal returns to the "READY" prompt.

```
BASIC 10.0 V0.98.911001 ALL RIGHTS RESERVED
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 $1.07870447 \times 10^{-3} = 0.00107870447$. 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, we promised a maze, so we 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 program 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 program so that it will run continuously.

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

If you **RUN** this program, it will start drawing a maze forever, that looks like the screenshot below. You can stop it at any time by pressing **RUN STOP**, or you can pause it by pressing **NO SCROLL**, and unpause it by pressing **NO SCROLL** again. If you press **RUN STOP**, 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:

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

READY.

RUN

```
?BREAK IN 10
READY.
```

That works nicely, and draws a very famous maze [1]. We can, however, make the program 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\$()** 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 program? This is quite normal. For complex programs, 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 we 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" program again. But this time, we 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 **\$N** to a random integer between 1 and 100. Once you have your guess ready, have a look what we came up with below. You might have made a different program 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 program much more fun to play. Can you beat it?

Exercises to try

1. Can you make the maze program make different mazes?

The maze program currently displays equal numbers of \square and \blacksquare . Can you change the program 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” program to choose a number between 1 and 10?

At the moment, the program picks a number between 1 and 100. Modify the program 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 **SN**, 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” program 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” program randomly choose between two different greeting messages when it starts.

The “guess my number” program 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.*

CHAPTER 9

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 Chapter/Appendix C on page C-3. 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:

```
AX = 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
```

```
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 PALINDROME")
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.

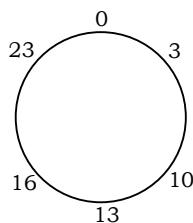
The cryptographic systems we describe here are very simple. Obviously, they shouldn't be used today because they are easily broken by techniques of cryptanalysis. Nevertheless, they illustrate some basic techniques and show how we might structure a sample program.

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
```

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.

10

CHAPTER

C64, C65 and MEGA65 Modes

- **Switching Modes from BASIC**
- **The KEY Register**
- **Accessing Memory from BASIC 65**
- **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.

By default, the MEGA65.ROM file boots to MEGA65-mode (including BASIC 65), and provides a method to switch to C64-mode via the `GO 64` command. However, it is also possible to use an original C65 ROM (version 91XXXX.BIN) renamed to MEGA65.ROM, making the MEGA65 start in C65-mode with BASIC 10. This also provides the same functionality to switch to C64-mode.

Therefore, dependent on your boot ROM choice, you have:

Boot Mode	ROM version	BASIC	C64-mode
MEGA65	92XXXX	BASIC 65	<code>GO 64</code>
C65	91XXXX	BASIC 10	<code>GO 64</code>

For readers familiar with the C128, the most important difference is that all of the MEGA65's new features can be accessed from every mode, and that you can even switch back and forth between the different modes. It's also possible to create hybrid modes that combine different features from the different modes – all you need is the **MAP** instruction and the KEY register address, which is **53295** (\$D02F).

This chapter explains the different modes, the **MAP** instruction, and the KEY register, which allows you to change the mode of operation of the MEGA65. This chapter also explains how to use BASIC commands to switch from one mode to another.

SWITCHING MODES FROM BASIC

The MEGA65 is used in either C64-mode (running BASIC 2), C65-mode (running BASIC 10) for ROM versions 91XXXX, or MEGA65-mode (running BASIC 65) for ROM versions 92XXXX.

However, various MEGA65 features can be accessed from all modes, and all MEGA65 features are available to programs written in assembly language / machine code. More information on how to write such programs can be found in the various appendices.

From MEGA65/C65 to C64-mode

To switch from MEGA65/C65 to C64-mode, use the familiar `GO 64` command, which is identical to switching to C64 mode on a C128:

GO 64

ARE YOU SURE? Y

Note that any programs in memory will be lost in the process of switching modes. This is the same as the C128. Alternatively, you can hold  down while pressing the reset button or switching the MEGA65 on. Again, this is the same as the C128.

From C64 to MEGA65/C65-mode

To switch from C64 to MEGA65/C65-mode, use the following command. **Note that this command does not ask you for confirmation!**

SYS 58552

Alternatively, you can switch back to MEGA65/C65-mode by pressing the reset button on the left-hand side of the MEGA65, or by switching the MEGA65 off and on again.

Another option is to press and hold  for between half to one second, and then press **F5** from the Freeze Menu. This simulates pressing the reset button.

Note that any programs in memory will be lost in the process of switching modes. This is the same as the C128.

Entering Machine Code Monitor Mode

The Machine Code Monitor can be entered by typing either the **MONITOR** command from BASIC 65/10, or by holding  down, and then pressing the reset button on the left-hand side of the MEGA65.

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 the time, software that was made for the C64 and VIC-II may inadvertently use these new registers, resulting in unexpected behaviour. Therefore, the creators of the C65 created a way to hide the extra VIC-III registers from old C64 programs. Enabling and disabling (or hiding and un-hiding) the extra registers is done via the KEY register. For more information about which registers are disabled and enabled in each of the VIC-II, VIC-III and VIC-IV I/O modes, refer to Chapter/Appendix F on page F-8.

The KEY register, located at address **53295**, is an unused register of the VIC-II, which you can **POKE** to and **PEEK** from, similar to other registers. But the KEY register has a special function: If you write two certain 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 MEGA65.

Exposing Extra C65 Registers

For example, to enable the VIC-III's new registers when in C64-mode, you must **POKE** the values 165 and 150 into the KEY register. The easiest way to do this is to switch your MEGA65 off and on again, and type GO 64 and answer Y to enter C64-mode.

Note: If you perform these POKEs while in C65-mode, the MEGA65 may not function correctly.

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

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

When you enter these commands, the MEGA65 returns a **READY.** prompt, and seemingly nothing else has happened. This is expected, because the MEGA65 has only enabled the VIC-III's new registers (and some other C65-mode features). The C64 BASIC and KERNAL will still function as normal, and it may appear that nothing has changed... But things have changed.

For example, you can do something that the C64 and its VIC-II can't do: smoothly change one colour to another. The VIC-III has registers that allow you to change the red, green and blue components of the colours. Now that the VIC-III registers are enabled, it's possible to 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 (\$D100 - \$D1FF). Blue is colour 6, so a change to register 53510 (53504 + 6, or \$D106) is required. An example BASIC listing that includes a **FOR** loop to change the colour is:

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

Once the program has been entered, type **RUN** on a new line. This will make the background of the screen fade from blue to purple. If you would like to make the effect progress faster, increase the 0.2 to a larger number such as 0.5. To make it slower, change it to a smaller number such as 0.02. You can also change the red component by **POKEing** a different number to 53504 - 53759 (\$D100 - \$D1FF), the green component at 53760 - 54015 (\$D200 - \$D2FF), or the blue component at 54016 - 54271 (\$D300 - \$D3FF). For example, to have the border and text (since they are both normally "light blue") fade from blue to green, you can try:

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

Disabling the C65/MEGA65 Extra Registers

You can also disable the VIC-III registers again by **POKE**ing any number into the KEY register, e.g.:

```
POKE 53295,0
```

If you **RUN** the examples above again, the colours won't change because the registers are disabled. Instead, writing to those addresses changes some of the VIC-II's registers, as on a C64 they appear several times over. Fortunately for the above example, the registers used have no obvious side-effects. This is because the modified registers in the examples above on a standard VIC-II are used to change the sprite positions. Since there are no sprites on the screen, you won't see anything change.

Enabling MEGA65 Extra Registers

The MEGA65 has *even more* registers than the C65. To enable these in C64-mode, it's required to **POKE** another two values into the KEY register:

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

Again, you won't see any immediate difference, which is similar to when enabling the VIC-III registers. However, now the MEGA65 can access not only the VIC-II and VIC-III registers, but also the VIC-IV registers. If you like, you can try the examples from earlier in this chapter to see that the VIC-III registers are accessible again. But now you can also do MEGA65 specific things. For example, if you wanted to move the start of the top border higher on the screen, you can try something such as:

```
POKE 53320,60
```

Alternatively, you can have some fun and animate the screen borders, by having them move closer and further apart:

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

The above example has the loop count backwards (from 255 to 0), so that you don't end up with only a tiny sliver of the text visible. You can make it go forwards if you like.

If you do get stuck with only a sliver of the screen, you can press **RUN STOP** and **RESTORE**.

You might be wondering: Why does **RUN STOP** and **RESTORE** work when these are VIC-IV registers that the C64-mode BASIC and KERNAL don't know about? The reason is the VIC-IV has a feature called "hot registers", where certain C64 and C65 registers cause some 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 (\$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 make the screen shake around instead 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 Chapter/Appendix [M](#) on page [M-5](#).

Traps to Look Out For

In all modes, 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 VIC-II mode whenever it is accessed. This means if you perform actions such as check the drive status, or **LOAD** or **SAVE** a file, the KEY register will be reset, and only the VIC-II registers will be enabled. You can of course enable the C65 or MEGA65 registers by **POKEing** the correct values to the KEY register again.

ACCESSING MEMORY FROM BASIC 65

BASIC 65 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, split into 6 banks of 64KB each. They are:

- BANK 0 and BANK 1 – acts as the C65's normal 128KB RAM.
- BANK 2 and BANK 3 – normally write-protected, and contains the C65's ROM image.
- BANK 4 and BANK 5 – used for all graphic routines in BASIC 65 for high resolution bitplane graphics. BASIC 10 doesn't use banks 4 and 5.

Using the **BANK**, **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, by using the **DMA** command, you can copy the current contents of the screen and colour RAM into BANK 4 with:

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

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

```
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
```

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 uses. It is used by the C65 ROM and BASIC 65 to manage what memory it can use at any particular point in time. For further explanation of the **MAP** instruction, refer to the relevant section of Chapter/Appendix [G](#) on page [G-8](#).

PART IV

SOUND AND GRAPHICS

11

CHAPTER

Graphics

Let's have some fun with graphics! In this part of the book, we want to examine the MEGA65's graphics modes by walking through example code in machine language to get to know the various options of the MEGA65 in the area of graphics. First of all, it is important to know that the MEGA65 supports three different basic graphics modes:

- Bitmap graphics
- Graphics based on character sets
- Bitplanes

BITMAP GRAPHICS

In bitmap graphics every pixel of a graphic is stored separately. The way the pixels are held in memory varies from system to system and in most cases depends on the performance of the hardware. If memory would be unlimited, the easiest way to remember a pixel is to save its RGB-values in three separate bytes. Example: 0xFFFFFFF for white would result in three values to be stored: \$FF, \$FF and \$FF. To be honest, this is too simple and not really efficient. Let's think about another way. Why not defining a color table (or color palette) and store the RGB values once and finally only reference the color by its index in the table? This will save us a lot of memory! Let's imagine we would create a colorful 8x8 bitmap to represent an "A" on the screen. Colourful means, we want it in some brownish colors. The color table for it may look like this:

Index	Color
0	Black
9	Brown
8	Orange
A	Light Red
F	Light Grey
7	Yellow

The color values, by the way, are exactly the same as the color values from the standard C64 color palette. Next we design the "A". Each pixel references a value from the color table above.

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0
1	A	F	7	F	A	8	0	0
2	F	F	0	0	8	A	0	0
3	7	7	0	0	8	A	0	0
4	7	F	F	A	A	A	0	0
5	F	A	0	0	8	A	0	0
6	A	A	0	0	A	A	0	0
7	8	9	0	0	8	A	0	0

But how much memory does this little graphic use?

If we create an one-dimensional array, we will get an array with 64 elements, because our graphic consists of 8x8 pixels = 64 color values that have to be saved. If we transfer that to the memory of the MEGA65, it means that we have 64 bytes to store in memory. However, full-screen graphics are made up of far more pixels. On the C64 and of course also on the MEGA65, 320x200 pixels are required to generate a graphic that fills the entire screen. If we transfer this to our array, we would have a total of 64,000 entries. Converted to the memory of the MEGA65, that is 64,000 bytes or nearly 64 kilobytes of data! If we now also consider that the good old C64 only had 64K of RAM available, we recognize the Drama! That's just too much data! We need strategies to reduce our bitmap data. On the C64 we had to two types of bitmap graphics and both come with its own concepts to use as less memory as possible.

- Hires
- Multicolor (MCM)

Hires

First, a bitmap is divided into blocks of 8x8 pixels each. In order to achieve the full resolution of 320x200 pixels, 40 of such blocks next to each other builds up a line. If we now build up 25 lines, we arrive at a graphic that fills the complete screen.

	1	2	3	4	5	6	7	8	9	0	1	1	1	1	1	1	1	1	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	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																																						

Splitting into blocks makes sense because this gives us the chance to reduce the data drastically. Each line of a 8x8 block are 8 bits, so why not forget the color indexes and just say each pixel set represents a "1" in the line and each pixel not set corresponds to "0".

	7	6	5	4	3	2	1	0	dec	hex
0	0	0	0	0	0	0	0	0	0	00
1	0	1	1	1	1	0	0	0	120	78
2	1	1	0	0	1	1	0	0	204	CC
3	1	1	0	0	1	1	0	0	204	CC
4	1	1	1	1	1	1	0	0	252	FC
5	1	1	0	0	1	1	0	0	204	CC
6	1	1	0	0	1	1	0	0	204	CC
7	1	1	0	0	1	1	0	0	204	CC

As shown above now you can easily convert the bits of each line to its hex value and you finally get 8 bytes of data. This is the central idea in hires graphics and with this concept you save a lot of memory. Here in this example it's 8 Bytes versus 64 Bytes if you have to manage all the color references. Let's count it up for a full screen picture: We have 40x25 Blocks, that is 1000 blocks in total. Each block is 8 Bytes or 1 Kilobyte, so we'll result in "only" 10K for a full screen picture.

This is a brilliant solution, but now have a problem: We lost the colors!

If you look at the first figure, it was very colorful. But do we really need so many colors? This is the second important concept in hires bitmap graphics: Less colors means less memory. In fact hires mode limits you to **two colors** per 8x8 pixel block. This information is stored in the Screen-RAM, not in the Color-RAM as one might assume. And again the idea why is really clever: In Screen-RAM you can hold values between \$00 and \$FF whereas in the Color-RAM it makes only sense to store values between \$0 and \$0F to reference one specific color from the C64 color palette which consists of 16 colors (\$0-\$F). On the MEGA65 you can have even more colors and you are able to tweak the default colors, but this will be explained later.

Back to the Screen-RAM. Here you store **two colors** for each 8x8 block. If you split the byte into its Hi-Byte and Low-Byte you have the chance to put a background and a foreground color into it! The value \$0a for example can be seen as \$0 and \$a. This way we'll get a black background and light red for the pixels in the block. In the end this means, a fullscreen hires bitmap will consume not 10 but 20 Kilobytes. 10K for the raw bitmap data as mentioned earlier and another 10K for the colors inside the Screen-RAM.

Programming simple Hires Bitmaps

Let's code!

PART V

HARDWARE

12

CHAPTER

Using Nexys4 boards as a MEGA65

- Building your own MEGA65 Compatible Computer
- Working Nexys4 Boards
- Power, Jumpers, Switches and Buttons
- Keyboard
- Preparing microSDHC card
- Loading the bitstream from QSPI
- Widget Board

- **PMOD-to-Joystick Adaptor**
- **Useful Tips**

BUILDING YOUR OWN MEGA65 COMPATIBLE COMPUTER

You can build your own MEGA65-compatible computer by using either a Nexys4DDR (aka. Nexys A7) or the older Nexys4 (Non-DDR) FPGA development boards. This appendix describes the process to set up a Nexys4DDR (Nexys A7) board for this purpose (which is the newer, preferred board). 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/NexysA7/Nexys4 boards.

WORKING NEXYS4 BOARDS

There are currently 3 Nexys FPGA boards which can be setup as a MEGA65:

The Nexys4 board

No longer manufactured but still available for sale on some websites with old stock.



Documentation:

- <https://reference.digilentinc.com/reference/programmable-logic/nexys-4/reference-manual>
- https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-4/nexys4_rm.pdf

The Nexys4DDR board

No longer manufactured but still available for sale on some websites with old stock.

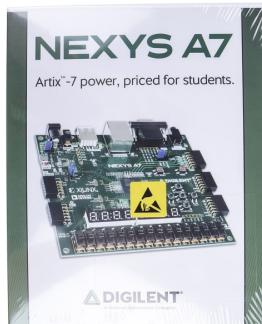


Documentation:

- <https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual>
- https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-4-ddr/nexys4ddr_rm.pdf

The Nexys A7

This is the re-branded version of the above Nexys4 DDR board:

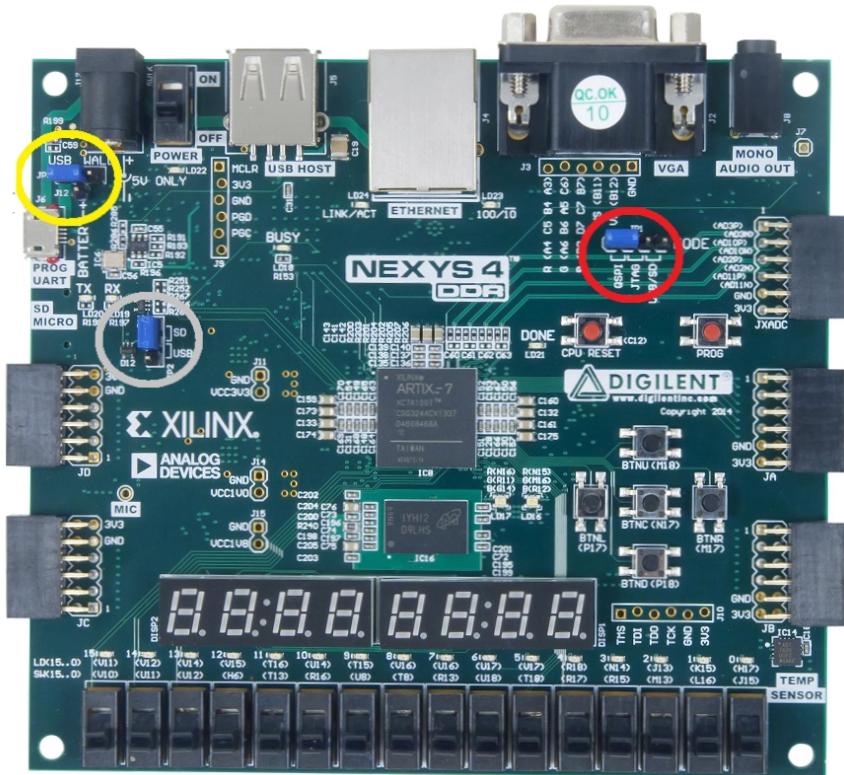


Documentation:

- <https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual>
- https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-a7/nexys-a7_rm.pdf

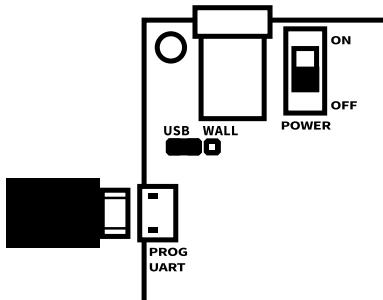
POWER, JUMPERS, SWITCHES AND BUTTONS

This top-down picture highlights the key jumper positions of interest on the Nexys4 board:



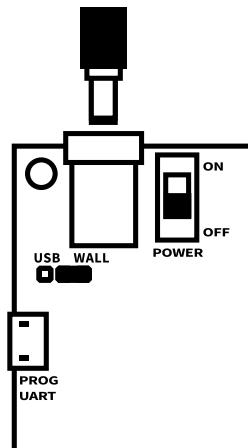
The Nexys4 boards can be powered in two ways: using an external power supply, or from a standard USB port.

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 Nexys4's micro-usb connector. Place the JP3 jumper on pins 1 and 2 to select USB power. Use the switch to power up the Nexys4.

External Power Supply



The MEGA65 core can consume a lot of power, and a standard USB port could potentially be too little for the Nexys4 board. In particular, writing to the SD card might hang or perform odd behaviour. Therefore you should consider a 5V power supply.

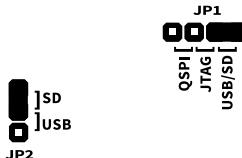
Digilent sell a power supply for the Nexys4 board, and we recommend you use this to ensure you avoid the risk of damage to your Nexys4 board. The chosen power supply should be center positive, 2.1mm internal diameter plug, and should deliver 4.5VDC to 5.5VDC rated at least 1 Amp.

Connect the power supply cable to the supply plug of the Nexys4. Place the JP3 jumper on pins 2 and 3 to select WALL power. Use the switch to power up the Nexys4.

Other Jumpers and Switches

For your initial set up, we'd suggest you set the following jumpers on your Nexys4 board to these positions:

- JP1 – USB/SD
- JP2 – SD

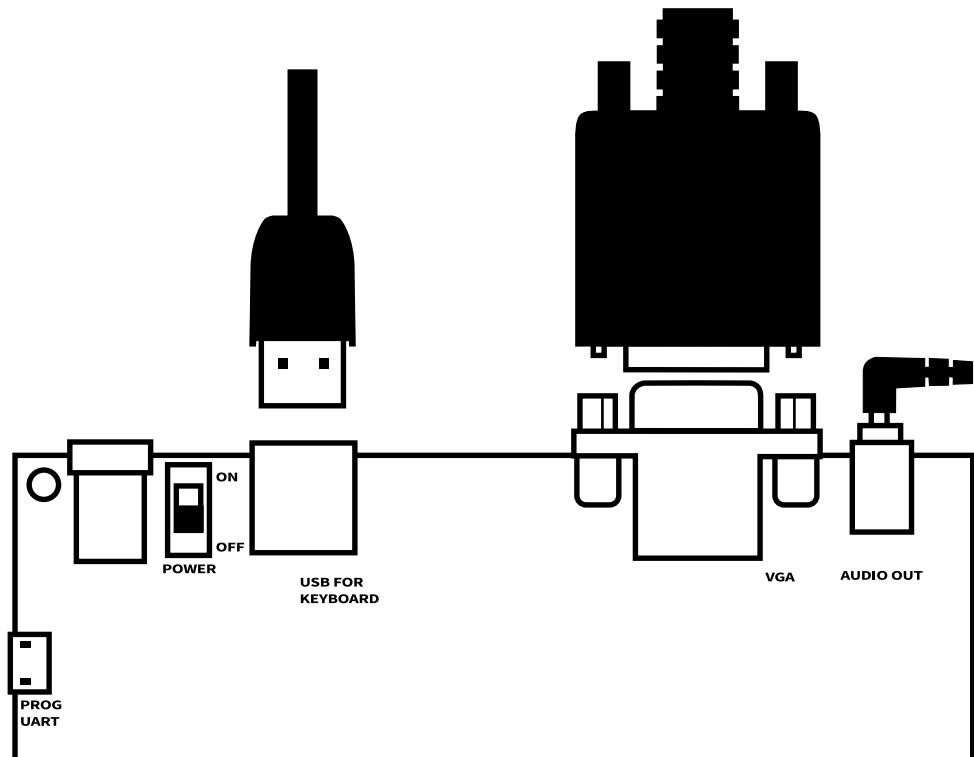


This will assure that the bitstream files will get loaded from your SD card on start-up.

At some later stage, you may prefer to load the bitstream from the on-board QSPI flash, and at that point, you can revisit your JP1 jumper setting and adjust it to the QSPI position.

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 Nexys4 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.

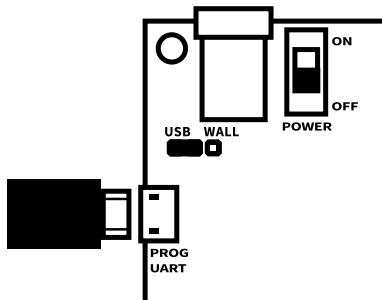
Communicating with your PC

There may be occasions where you wish to communicate with your Nexys4 board from your PC, in order to perform activities such as:

- Flash your QSPI flash chip via Vivado
- Upload bitstream files directly from your PC (via m65 tool)

- Make use of support tools such as M65Connect, m65, mega65_ftp, m65dbg, etc

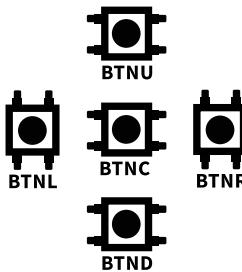
On such occasions, you will need to connect your micro-usb cable up to your PC.



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: BTND acts as though you have pressed **RESTORE**, 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.

To help you locate what the various MEGA65 keys are mapped to, the MEGA65 has a built-in virtual keyboard test feature. This can be accessed in two ways.

The easiest way is to keep **ALT** held down in while switching on the Nexys4, or resetting the Nexys4 with the "PROG" button. The configure menu will be presented and by pressing 3, the virtual keyboard will be presented on a black background.

RUN STOP	ESC		ALT	ASC DIN	NO SCRL	F1 F2		F3 F4	F5 F6	F7 F8	F9 F10		F11 F12	F13 F14	HELP	
~ —	!	"	#	\$	%	&	7	()	'	0	+	-	GBP	CLR HOME	INST DEL
TAB	Q	W	E	R	T	Y	U	I	O	P	@	*	PI ~^	RESTORE		
CTRL	SHFT LOCK	A	S	D	F	G	H	J	K	L	{:	}	=	RETURN		
M=	SHIFT	Z	X	C	V	B	N	M	<	>	?	/	SHIFT	UP		
SPACE													LEFT	DOWN	RGHT	

Pressing a key on the USB keyboard will show the highlighted key on the virtual keyboard to help you identify the key mapping.

The other way to access the virtual keyboard is from within the MEGA65. Hold **M** and press **TAB** to access the Matrix Mode Debugger. From here, enter the following:

```
s ffd3615 ff
```

This will open a semi-transparent virtual keyboard at the top of the screen. Alternatively:

```
s ffd3615 ff ff
```

This will open a semi-transparent virtual keyboard in the centre of the screen.

Hold **M** and press **TAB** to exit Matrix Mode Debugger and return to the MEGA65.

Some key mappings with a USB keyboard

[RESTORE] is mapped to the PAGE UP key.

[RUN STOP] is mapped to **[ESC]**.

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.

Preparation steps for the Nexys4 board's SD card share much in common with the steps needed for real MEGA65 hardware, and as such, it is worth having a look over the [Configuring your MEGA65](#) chapter if you ever need details.

So in this section, we'll provide more details on the distinctive steps, and be more brief on the common steps.

One point of distinction between the Nexys board and the real MEGA65 hardware is that the latter already has a default bitstream/core provided, which permits you to format your SD card in the specific style required by the MEGA65.

For Nexys4 board owners however, you have no such default bitstream, so see [Bit-stream files](#) for more details on where the appropriate "nexys4.bit" or "nexys4ddr-widget.bit" files for your device can be downloaded from.

Preparation Steps

The steps are:

- Format the SD card in a convenient computer using the FAT32 file-system. The MEGA65 and Nexys4 boards do not understand other file systems, especially the exFAT file system.
- Copy your bitstream 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 Nexys4 board.
- Switch on the Nexys4 board.
- Enter the Utility Menu by holding  down on the USB keyboard you have connected to the Nexys4 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).
 - The other partition with type \$0C = VFAT32, where KERNAL, 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 Nexys4 board and remove the microSDHC card from the Nexys board and put it back into your PC
 - This time, copy the following items onto the SD card:
 - The bitstream file
 - The extracted files from within either the "**SD essentials.rar**" or "**SD essentialsNoROM.rar**" file that you downloaded from the MEGA65 filehost. (See [Installing ROM and Other Support Files](#) for more details).
 - If you have sourced your own preferred ROM file (e.g. "**911001.BIN**"), copy it onto the SD card also, and rename it to "**MEGA65.ROM**" (uppercase is essential).
 - Any .D81 disk image files you wish to make use of.
 - * Note that if a file named MEGA65.D81 is added to the SD card, it will be mounted automatically on startup.
 - * Make sure that all .D81 files 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 Nexys4 board.
 - Power the Nexys4 board back on. The MEGA65 should boot within 15 seconds.
 - On first start up, you will find yourself at the on-boarding screen, of which more details can be found in the [Configuring your MEGA65](#) chapter.

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

Please note that the above method of copying the bitstream file to the SD card means that the bitstream is loaded into the Nexys FPGA each time on boot - which takes around 13 seconds for the system to start. The bitstream can also be flashed using Vivado software into the QSPI flash to deliver a boot up time of 0.3 seconds.

For more detailed information on preparing and configuring your MEGA65, please refer to the [Configuring your MEGA65](#) chapter.

LOADING THE BITSTREAM FROM QSPI

While loading the bitstream from the SD card is the suggested (and well-trodden) path this document has chosen, of late, more nexys4 users have been exploring the alternative pathway of loading the bitstream from the QSPI flash. Some potential reasons they have chosen this pathway are:

- Faster loading times (0.3 seconds versus 13 seconds)
- Some people were interested in the possibility of flashing multiple cores onto their QSPI (via steps described in the [Cores and Flashing Chapter](#))
- Some people have experienced niggling issues with the SD card pathway, such as:
 - System unable to reboot from on-boarding screen
 - System unable to reboot from freeze-menu after switching between PAL/NTSC

In time, if this proves to be a more popular pathway, we can revise our documentation here to suit it. Here are some steps in brief.

Preparation Steps

For users that want to try this pathway, you will need to adjust the JP1 jumper setting to use QSPI and then follow the steps in the [Flashing the FPGAs and CPLDs in the MEGA65 chapter](#) in relation to [Installing Vivado](#) and [Flashing the main FPGA using Vivado](#).

Be forewarned that the installation of Vivado is a lengthy process (both in terms of download time, and installation time).

Once you have flashed Slot0 of your QSPI chip via Vivado, you can then follow the steps described in [Configuring your MEGA65](#) to perform the custom SD card formatting, installing of ROM and support files and on-boarding.

WIDGET BOARD

For Nexys board owners (all models), you may be interested in adding a widget board to your nexys device, in order to allow you to connect to:

- a genuine C64 or C65 keyboard
- 2 DB-9 joysticks

- Paddles or a 1351 Mouse
- Cartridges (not functioning as yet)

The widget board connects to your nexys board via its PMOD connectors.

It presently is only available as an unpopulated PCB, purchasable from here:

- https://oshpark.com/shared_projects/Y37xg9N7

The firmware, pcb diagram, schematic diagram and bill of materials can be found within the following github project:

- <https://github.com/sy2002/DM65PIC>

You will need to purchases parts from the bill of materials separately and populate the board yourself.

You may find this forum64.de thread of interest, if you would like to read on the experiences of others that undertook this process, or if you have questions of your own to ask:

- <https://www.forum64.de/index.php?thread/90465-mega-65-ports-add-on-card-still-available-if-where>

The pcb (c65Keyb.brd) and schematic (c65Keyb.sch) files can be found within the 'eagle/' subfolder of the DM65PIC project, and can be viewed via the free tool 'AutoDesks Eagle', available here:

- <https://www.autodesk.com/products/eagle/free-download>

Here are some photos of the widget board in use:





For convenience, the pcb and schematics diagrams for the widget board have also been provided in Chapter/Appendix W on page W-51.

Some additional backstory notes for the board are:

- The widget board is originally meant to sit inside a c65 case
- Expansion port is not usable and a bit needs to be cut out for c64 case
- Would be reasonable to adapt it for c64 case use (which definitely will be the more regular use)

PMOD-TO-JOYSTICK ADAPTOR

As an alternate (and cheaper) option for those that just want to add a DB9 joystick port via the PMOD connectors, a user from the community, TheChief, has devised a means to do so. More information can be found in the following Discord thread:

- [https://discord.com/channels/719326990221574164/
903079038015389716/929369283119685643](https://discord.com/channels/719326990221574164/903079038015389716/929369283119685643)



Joystick action	PMOD-JA Connections	DB9 Connections
Fire	jahi(9)	pin 6
Up	jalo(1)	pin 1
Left	jalo(2)	pin 3
Down	jahi(7)	pin 2
Right	jahi(8)	pin 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  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.
- 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 `Y58552` in C64-mode to switch to C65-mode.
- Type `G064` in C65-mode and confirm, by pressing `Y`, to switch to C64-mode, which is the same as 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  +  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 & Art welcomes your feedback, suggestions and contributions to this open-source digital heritage preservation project.

PART VI

CROSS-PLATFORM DEVELOPMENT TOOLS

13

CHAPTER

Emulators

- Using The Xmega65 Emulator
- Using the Live ISO image

At the time of writing, there is only one emulator for the MEGA65, **xmega65**; LGB's Xemu emulator suite. The LGB developers work hard to keep up with the development of the MEGA65; however, some MEGA65 emulation may not be accurate but should be sufficient for software development on the MEGA65.

During development, frequently test software on real hardware, such as a MEGA65 or FPGA board capable of running a MEGA65 core.

Download the MEGA65 emulator source code from <https://github.com/lgb1gblgb/xemu>.

Download pre-compiled versions from <https://github.lgb.hu/xemu/>.

A live ISO image containing the emulator, documentation, and other tools is available from Forum64.de at <https://www.forum64.de/index.php?thread/104698-xemu-live-system-iso-file/&postID=1549927#post1549936>.

USING THE XMEGA65 EMULATOR

USING THE LIVE ISO IMAGE

The Live ISO image is the product of a volunteer community; not the MEGA65 team. We include it for your convenience.

Creating a Bootable USB stick or DVD

There are many ways to create a live ISO image. The method you choose depends on your operating system and whether you wish to install to a USB drive or burn it to a DVD. Burning to a DVD is straightforward, assuming you own a computer that has a DVD writer. If you wish to create a faster bootable USB drive, try one of the methods below:

If you are using Windows, consider a tool like <http://www.isotousb.com/>.

On Linux, you can use the instructions at <https://fossbytes.com/create-bootable-usb-media-from-iso-ubuntu/>.

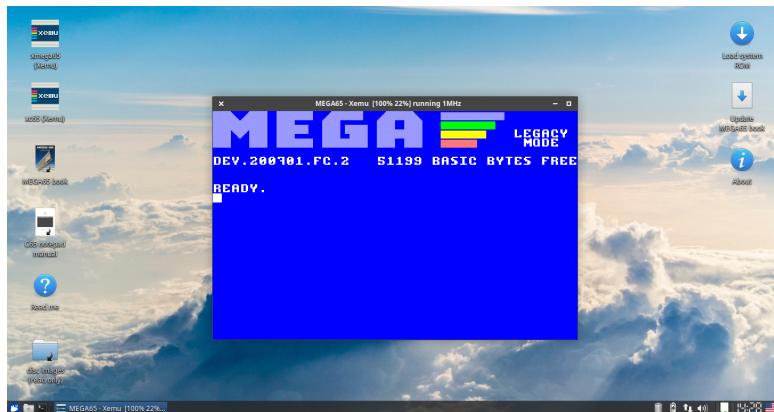
For Apple Macs, consider these instructions at <https://ubuntu.com/tutorials/create-a-usb-stick-on-macos#1-overview>.

Similar instructions are available for other popular computers, such as Amigas (<https://forum.hyperion-entertainment.com/viewtopic.php?t=3857>), or Sun UltraSPARC workstations (<https://forums.servethehome.com/index.php?threads/how-to-create-a-bootable-solaris-11-usb.1998/>).

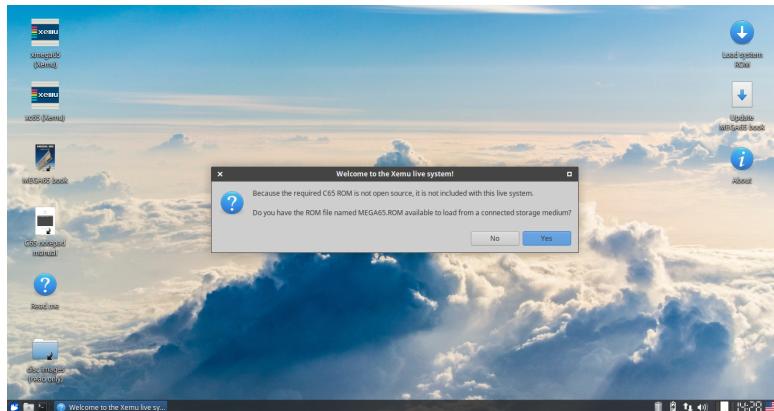
Finally, the popular, easy-to-use, and free cross-platform belanaEtcher is available at <https://www.balena.io/etcher/>.

Getting Started

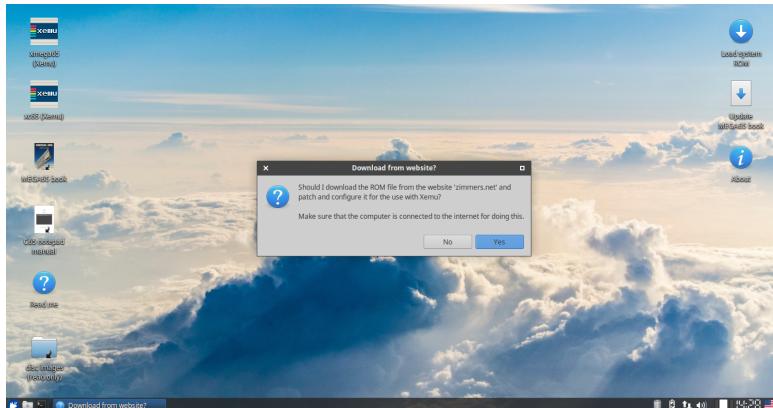
To avoid potential copyright issues, the bootable ISO image does not include proprietary ROMs for the MEGA65; such as legacy C65 ROMs. It does include an open-source replacement ROM from our OpenROMs project. This ROM will boot into a BASIC 2 environment that you can use to load and execute many C64 programs as shown in the image below:



If you wish to use a C65 ROM that includes BASIC 10, download the appropriate ROM file and place it on another USB stick named **MEGA65.ROM**. On start-up, the MEGA65 will ask if a ROM has been downloaded; as shown in the image below:



If the Live ISO cannot find a ROM, it will prompt you to download a ROM; as shown below:



Other Features of the Live ISO

As the previous screen-shots show, the Live ISO provides various and convenient desktop shortcuts. On the left-hand side, there are shortcuts for launching the MEGA65 emulator and the C65 emulator so you can test that programs will run on both platforms. As previously mentioned, both emulators are a work in progress and may not be 100% compatible.

Another link provides access to the MEGA65 Book. This all-in-one volume, of approximately 800 pages, contains the official MEGA65 documentation. The majority of this developer's guide is also present in the MEGA65 Book.

This ISO also includes documentation for the C65 Notepad; a program for the C65 and MEGA65 written by Snoopy (the developer of the Live ISO image). A "read me" file contains further information about the Live ISO.

Finally, on the right-hand side, there are links to download a C65 ROM and to update the MEGA65 Book to the latest version. This will ensure you don't need to create a new bootable image each time a frequent update is made to the MEGA65 Book.

To access all contents of the Live ISO image, use the file explorer.

14

CHAPTER

Data Transfer and Debugging Tools

- **m65 command line tool**
- **M65Connect**
- **mega65_ftp**
- **TFTP Server**
- **Converting a BASIC text file listing into a PRG file**

The key to effective cross-platform development is having quick and easy means to deploy and test software on the MEGA65. This is especially true while the MEGA65 emulator continues to be developed. In fact, even once the MEGA65 emulator is complete, it is unlikely that it will be able to offer full compatibility at full speed, because the MEGA65 is much more demanding to emulate than the C64.

There are a variety of tools that can be used for data transfer and debugging. These typically function using either the MEGA65's serial monitor interface, or via the MEGA65's fast ethernet adapter. The serial monitor interface is available via the UART lines on the JB1 header.

If you do not have access to the serial monitor interface, there are tools being developed for the fast ethernet port that provide some, but not all, of the capabilities of the serial monitor interface. These will be documented as they become available. The remainder of this chapter focusses on methods that access the serial monitor interface.

You can either connect a 3.3V UART adaptor to the appropriate lines, or more conveniently, connect a TE-0790-03 JTAG debug module onto this connector. This gives you a USB connection that can be used for injecting software, remote debugging and memory inspection, as well as activating or flashing bitstreams. With this connection, there are the following tools:

M65 COMMAND LINE TOOL

The <https://github.com/mega65/mega65-tools> repository contains a number of tools, utilities and example programs. These tools are mainly for Linux but can be used on Windows with Cygwin. One of those is the **m65** command line tool. This is rather a swiss-army knife collection of utilities in one. Common useful functions include:

Screenshots using m65 tool

To take a screenshot of the MEGA65 use:

```
m65 -s
```

This will create a file called **mega65-screeen-000000.png**, or if that file already exists, the first non-used number will be used in place of **000000**.

Note that this screenshot function works by having **m65** emulate the function of the VIC-IV. Thus while it produces excellent looking digital screenshots, it may not exactly match the real display of the MEGA65. At the time of writing it does not render sprites or bitplanes, only text and bitmap-based video modes.

Load and run a program on the MEGA65

To load and run a program on the MEGA65, you can use a command like:

```
m65 -F -4 -r foo.prg
```

The **-F** option tells **m65** to reset the MEGA65 before loading the program.

The **-4** option tells **m65** to switch the MEGA65 to C64-mode before loading the program. If this is left off, then it will attempt to load the program in C65-mode.

The **-r** option tells **m65** to run the program immediately after loading.

Note that this command works using the normal BASIC LOAD command, and is thus limited to loading programs into the lower 64KB of RAM

Reconfigure the FPGA to run a different bitstream

To try out a different MEGA65 bitstream, a command like the following can be used:

```
m65 -b bitstream.bit
```

This will cause the named bitstream to be sent to the FPGA. As the FPGA will be re-configured by this action, and program currently running will not merely be stopped, but also main memory will be cleared. For models of the MEGA65 that are fitted with 8MB or 16MB of expansion memory, those expansion memories are implemented in external chips, and so the contents of them will not be erased.

For non-MEGA65 bitstreams (such as zxunomega65 and gbc4mega65), use the '**-q**' argument instead:

```
m65 -q bitstream.bit
```

Remote keyboard entry

The MEGA65's keyboard interface logic supports the injection of synthetic key events using the registers \$D615 – \$D617. The **m65** utility uses this to allow remote typing on the MEGA65 in a way that is transparent to software. There are three ways to use this:

```
m65 -t sometext
```

This form types the supplied text, in this case *sometext*, but does not simulate pressing **RETURN**. If you wish to simulate the pressing of **RETURN**, use **-T** instead of **-t**, e.g.:

```
m65 -T list
```

This would cause the LIST command to be typed and executed.

Finally, it is possible to begin general remote keyboard control via:

```
m65 -t -
```

In this mode, any key pressed on the keyboard of the computer where **m65** is running will be relayed to the MEGA65. Note that not all special keys are supported, and that there is some latency, so using key repeat can cause unexpected results. But for general remote control, it is a very helpful facility.

Unit testing and logging support

The **m65** tool includes support to facilitate remote unit testing directly on MEGA65 hardware. When *unit testing mode* is active, **m65** waits for the MEGA65 to send certain byte sequences over the serial interface which signal the current state (started, passed, failed) of a given test. Additionally, it is possible to send log messages from the MEGA65 to the host computer.

Unit testing mode is entered by calling **m65** with the **-u** flag. To run a remote BASIC program in C65-mode and simultaneously put **m65** into *unit testing mode*, the following command can be used:

```
m65 -Fur attic-ram.prg -w tests.log
```

The **-F** and **-r** options tell **m65** to reset the MEGA65 before loading the program "attic-ram.prg" and then automatically run it. The **-u** option then tells **m65** go into *unit testing mode* instead of exiting after launching the program. The optional **-w** option makes **m65** append the test results to the file "test.log" (creating the file if it doesn't exist).

Please note that **m65** automatically exits from *unit testing mode* if no test state signals were received for over 10 seconds.

Support is provided for sending unit test signals to the host computer from C and BASIC 65 programs:

Using unit tests with C

The MEGA65 libc contains support for unit testing via functions defined in **tests.h** and **tests.c**.

To signal the start of a test, include **tests.h** and use

```
unit_test_setup("testName",issueNumber);
```

where "*testName*" is a human-readable name of the test (e.g. "VIC-II") and *issueNumber* a reference to the corresponding bug issue (for example, the issue number from github).

After starting a test, it's possible to signal passed tests with the `unit_test_ok()` function:

```
unit_test_ok();
```

A failed test is signalled with `unit_test_fail()`:

```
unit_test_fail("fail message");
```

Each time the `unit_test_ok()` or `unit_test_fail()` functions are called, the *sub issue* of the test (reported on the host computer) is incremented. This makes it easier to combine and identify multiple tests in one file.

You can send arbitrary log messages via `unit_test_log()`:

```
unit_test_log("hello world from mega65!");
```

...and finally, when all is done, the end of unit testing is signalled by the use of

```
unit_test_done();
```

Using unit tests with BASIC 65

b65support.bin is a machine language module providing support for unit testing from BASIC 65, available in the **bin65** folder of the mega65-tools repository. This module works by redirecting the USR vector to perform the functions needed to communicate with the testing host.

In an automated test scenario, you may want to inject the **b65support.bin** binary into MEGA65 RAM by using **m65**:

```
m65 -C mega65-tools/bin65/b65support.bin@15fe
```

Of course it's also possible to load **b65support.bin** directly from the MEGA65 by mounting the **M65UTILS.D81** image from the freezer and issuing

```
BLOAD "B65SUPPORT.BIN"
```

After loading, **b65support.bin** is initialized with

```
SYS $1600
```

Once initialized, the following functions are provided by **b65support.bin**:

```
A=USR({<issueNum>})
```

prepares a new test with number <issueNum> and resets subissue number to 0

```
A=USR("={<testName>}")
```

sets test name and sends test start signal; for example: **A=USR("=VIC-III")** sets the test name to 'VIC-III' and signals the host computer that the test has started.

```
A=USR("//<logMessage>")
```

sends a log message to the host computer

```
A=USR("P")
```

sends the 'passed' signal to the host computer and increases the sub issue number

```
A=USR("F")
```

sends the 'test failed' signal to the host computer and increases the sub issue number

```
A=USR("D")
```

sends the 'test done' signal to the host computer

All calls return the current sub issue number or **?ILLEGAL QUANTITY ERROR** in case of calling an invalid command.

BASIC 65 example

The following is a complete BASIC 65 example showing how to use **m65**'s unit testing features:

```

100 rem attic ram cache test
110 poke $bfffff2,$e0          : rem enable attic ram cache
120 sys $1600                  : rem init test module
130 a=usr(379)                 : rem set issue number
140 a=usr("=attic-ram-cache") : rem set test name
150 bank128:poke0,65          : rem just to be sure
160 b0=$8000000 : bi=$8000100 : rem attic ram areas to be tested
170 for r=0 to $ff
180   poke b0+r,0               : rem fill area 1 with 0
190   poke bi+r,$ff             : rem fill area 2 with $ff
200 next r
210 for t=i to 10              : rem 10 tries
220   poke b0,32                : rem write to b0
230   for x=0 to $ff:ti=bi+x
240     a=peek(b0)              : rem read from b0
250     b=peek(ti):b=peek(ti) : rem read twice from ti
260     ifb<>255 then f=t:t=11:x=256 : rem this shouldn't happen
270   next x
280 next t
290 if f=0 then begin
300   print "no faults detected after";t;"tries."
310   a=usr("p")                : rem signal 'test passed' to host
320 bend : else begin
330   a=usr("f")                : rem signal 'test failed' to host
340   print "hyper ram fault detected after";f;"tries."
350   print "peek($";hex$(ti);") [ti] is";b;"but should be 255"
360 bend
370 a=usr("d")                  : rem test done

```

M65CONNECT

This is a cross-platform graphical tool available for Windows, Linux and MacOSX, which allows access to most of the functions of the **m65** command-line tool, without needing to use a command line, or being able to compile the tool for your preferred operating system.

The repository for M65Connect is: <https://github.com/MEGA65/m65connect>

The latest binary version is available from <https://files.mega65.org>.

With the MEGA65 or Nexys FPGA switched off, connect a USB cable from your computer to the MEGA65 or Nexys FPGA board. Run the *M65Connect* executable and follow the prompts to connect. The program will help you identify which USB Serial Port to communicate over.

With this tool you can easily transfer PRG programs and a variety of other files. *M65Connect* can handle the transfer, switching to C64-mode, and execution of programs.

MEGA65_FTP

The **mega65_ftp** utility from the <https://github.com/mega65/mega65-tools> repository is a little misleadingly named: While it is a File Transfer Program, it does not use the File Transfer Protocol (FTP). Rather, it uses the serial monitor interface to take remote control of a MEGA65, and directly access its SD card to enable copying of files between the MEGA65 and the host computer.

Note that it does not perfectly restore the MEGA65's state on exit, and thus should only be used when the MEGA65 is at the READY prompt, so that any running software doesn't go haywire. In particular, you should avoid using it when a sensitive program is running, such as the Freeze Menu, MEGA65 Configuration Utility, or the MEGA65 Format/FDISK utility. (This problem could be solved with a little effort, if someone has the time and interest to fix it).

When run, it provides an FTP-like interface that supports the **get**, **put**, **rename** and **dir** commands. Note that when putting a file, you should make sure that it is given a name that is all capitals and has a DOS-compatible 8.3 character file name. This is due to limitations in both **mega65_ftp** and the MEGA65's Hypervisor's VFAT32 file system code. Again, these problems could be fixed with a modest amount of effort on the part of a motivated member of the community.

Finally, the **mega65_ftp** program is very slow to push new files to the MEGA65, typically yielding speeds of around 5KB/sec. This is partly because the serial monitor interface is capable of transferring data at only 40KB/sec (when set to 4,000,000 bits per second), and partly because the remote control process results in a lot of round-trips where helper routines are executed on the MEGA65 to read, write and verify sectors on the SD card. It would be quite feasible to improve this to reach close to 40KB/sec, and potentially faster using either some combination of data compression, de-duplication of identical sectors (especially when uploading disk images) and other techniques. Again, this would be a very welcome contribution that someone in the community could contribute to everyone's benefit.

TFTP SERVER

Work on a true TFTP server for the MEGA65 that supports fast TFTP transfers over the 100mbit ethernet has begun, and can be used to very quickly read files from the MEGA65. Speeds of close to 1MB/sec are possible, depending on SD card performance. Rather than using DHCP, this utility will respond to *any* IP address that ends in .65. It always uses the MAC address 40:40:40:40:40:40. True DHCP support as well as using the MEGA65's configured ethernet MAC address may be added in the future.

More importantly, support for writing files to the SD card is not yet complete, and is blocked by the need for the implementation of the necessary functions in the MEGA65's Hypervisor for creating and growing files. A particular challenge is enabling the creation of files with contiguous clusters as is required for D81 disk images: If a D81 file is fragmented, then it cannot be mounted, because the mounting mechanism requires a pointer to the contiguous block of the SD card containing the disk image. In the interim, **mega65_ftp** can be used as a substitute.

CONVERTING A BASIC TEXT FILE LISTING INTO A PRG FILE

If you have a untokenised BASIC program in plain text format sourced from somewhere like an internet post, and you wish to try it on the MEGA65 without typing it in, it is possible to convert it to a PRG.

C64List is a Windows-based command-line tool that will allow you to make the conversion. Once you have a .PRG file, you can use a tool like M65Connect to upload it to the MEGA65 or Nexys FPGA.

C64List is available for download from <http://www.commodoreserver.com/Downloads.asp>

Ensure you have a program listing saved to a file on your local computer (for example, *program.txt*) encoded as ANSI or UTF8.

Use C64List to convert the file to a PRG file using:

```
C64List program.txt -prg
```

Now you can upload your newly converted program to the MEGA65 with M65Connect or one of the other tools described previously.

It is worth noting that this method will not be 100% effective on listings with special PETSCII characters. Programs with PETSCII will require some editing on the MEGA65 itself before saving to disk.

15

CHAPTER

Assemblers

The table below shows an overview of assemblers known to work with MEGA65. For general use we recommend **ACME** as it has good support for the 45GS02 instruction set; is open source; and finally written in C. The latter means that it may be ported to run natively on the MEGA65 in the future.

Name	45GS02	Source	Reference
ACME	yes	C	https://sourceforge.net/projects/acme-crossass
KickAss	yes	Java	
Ophis	yes	Python	https://github.com/michaelcmartin/Ophis
BSA	yes	C	https://github.com/Edilbert/BSA
CA65	no ¹	C	https://github.com/mega65/cc65

The **BSA** assembler is currently used to build the **MEGA65.ROM**. Most of this source code is written in the syntax of the ancient **BSO** assembler (Boston Systems Office), which was used in the years 1989 - 1991 by software developers, working on the C65. The **BSA** Assembler has a compatibility mode, which makes it possible to assemble these old source codes with minor or none modifications. The **BSA** Assembler has currently only a description of commands embedded in the C-source of the assembler.

¹ Our fork of CA65 (part of CC65) correctly detects the MEGA65's CPU, but has no explicit support for the processor's features

16

CHAPTER

C and C-Like Compilers

- MEGA65 libc

Short answer: CC65 and KickC both work on the MEGA65.

Both CC65 and KickC are known to work on the MEGA65. However, both by default have only a C64 memory model, and use only 6502 opcodes. It would be super for someone to create a C65 memory configuration for CC65, and should not be too hard to do.

CC65 supports overlays, which could be powerfully used with the MEGA65's extra memory to allow programs larger than 64KB. However, this would require writing a suitable loader for such programs, which also does not yet exist.

Similarly, modifying the code generator of CC65 to use 45GS02 features would not be particularly difficult to do, and would help to overcome the otherwise horribly slow and bloated code that CC65 produces. Also adding first-class support for the 45GS02 CPU features in CA65 (or perhaps even better, making CC65 produce ACME compatible assembly output) would be of tremendous advantage, and not particularly hard to do. These would all be great tasks to tackle while you wait for your MEGA65 DevKit to arrive!

An example template for a C program that can be compiled using CC65 and executed on the MEGA65 can be found in the repository <https://github.com/MEGA65/hello-world>. This repository will even download and compile CC65, if you don't already have it installed on your system. This repository should work on Linux and Mac, and on Windows under the Windows Subsystem for Linux (WSL).

The LLVM compiler for MOS (<https://llvm-mos.org>) also has a rudimentary MEGA65 target that has been tested with C99, C++11, and Rust. The LLVM MOS compiler has experimental support for 65CE02 instructions (`-mcpu=mosw65ce02`) and development is ongoing.

MEGA65 LIBC

A C library is being developed for the MEGA65, and which already includes a number of useful features. This library is available from <http://github.com/mega65/mega65-libc>. The procedures, functions and definitions it provides are documented in a separate chapter.

The MEGA65 libc is currently available only for CC65, although we would welcome someone maintaining a KickC port of it.

17

CHAPTER

MEGA65 Standard C Library

- Structure and Usage
- `conio.h`

A C library is being developed for the MEGA65, and which already includes a number of useful features. This library is available from <http://github.com/mega65/mega65-libc>. The procedures, functions and definitions it provides are documented in a separate chapter.

The MEGA65 libc is currently available only for CC65, although we would welcome someone maintaining a KickC port of it.

STRUCTURE AND USAGE

The MEGA65 libc is purposely provided in source-form only, and with groups of functions in separate files, and with separate header files for including. The idea is that you include only the header files that you require, and add only the source files required to the list of source files of the program you are compiling. This avoids the risk of the compiler including functions in your compiled program that are never used, and thus wasting precious memory space.

Note that some library source files are written in C, and thus are present as files with a `.c` extension, while others are written in assembly language either for efficiency or out of necessity, and have a `.s` extension.

Typical usage is to either have the mega65-libc source code checked out in an adjacent directory, or within the source directory of your own project. In the latter case, this can be done using the git submodule facility.

The following sections document each of the header files and the corresponding functions that they provide.

CONIO.H

conionit

Description: Initialises the library internal state

Syntax: `void conioinit(void)`

Notes: This must be called before using any conio library function.

setscreenaddr

Description: Sets the screen RAM start address

Syntax: `void setscreenaddr(long addr);`

Parameters: `addr`: The address to set as start of screen RAM

Notes: No bounds check is performed on the selected address

getscreenaddr

Description: Returns the screen RAM start address

Syntax: `long getscreenaddr(void);`

Return Value: The current screen RAM address start address.

setcolramoffset

Description: Sets the color RAM start offset value

Syntax: `void setcolramoffset(long offset);`

Parameters: `addr`: The offset from the beginning of the color RAM address (\$FF80000)

Notes: No bounds check is performed on the resulting address. Do not exceed the available Color RAM size

getcolramoffset

Description: Returns the color RAM start offset value

Syntax: `long getcolramoffset(void);`

Return Value: The current color RAM start offset value.

setcharsetaddr

Description: Sets the character set start address

Syntax: `void setcharsetaddr(long addr);`

Parameters: `addr`: The address to set as start of character set

Notes: No bounds check is performed on the selected address

getcharsetaddr

Description: Returns the current character set start address

Syntax: `long getscreenaddr(void);`

Return Value: The current character set start address.

clrscr

Description: Clear the text screen.

Syntax: `void clrscr(void)`

Notes: Color RAM will be cleared with current text color

getscreensize

Description: Returns the dimensions of the text screen

Syntax: `void getscreensize(unsigned char* width, unsigned char* height)`

Parameters: **width:** Pointer to location where width will be returned

height: Pointer to location where height will be returned

setscreensize

Description: Sets the dimensions of the text screen

Syntax: `void setscreensize(unsigned char width, unsigned char height)`

Parameters: **width:** The width in columns (40 or 80)

height: The height in rows (25 or 50)

Notes: Currently only 40/80 and 25/50 are accepted. Other values are ignored.

set16bitcharmode

Description: Sets or clear the 16-bit character mode

Syntax: `void set16bitcharmode(unsigned char f)`

Parameters: `f`: Set true to set the 16-bit character mode

Notes: This will trigger a video parameter reset if HOTREG is ENABLED. See sethotregs function.

sethotregs

Description: Sets or clear the hot-register behavior of the VIC-IV chip.

Syntax: `void set16bitcharmode(unsigned char f)`

Parameters: `f`: Set true to enable the hotreg behavior

Notes: When this mode is ENABLED a video mode reset will be triggered when touching \$D011, \$D016, \$D018, \$D031 or the VIC-II bank bits of \$DD00.

setextendedattrib

Description: Sets or clear the VIC-III extended attributes mode to support blink, underline, bold and highlight.

Syntax: `void setextendedattrib(unsigned char f)`

Parameters: `f`: Set true to set the extended attributes mode

togglecase

Description: Set lower case character set

Syntax: `void setlowercase(void)`

togglecase

Description: Set upper case character set

Syntax: `void setuppercase(void)`

togglecase

Description: Toggle the current character set case

Syntax: `void togglecase(void)`

bordercolor

Description: Sets the current border color

Syntax: `void bordercolor(unsigned char c)`

Parameters: `c`: The color to set

bgcolor

Description: Sets the current screen (background) color

Syntax: `void bgcolor(unsigned char c)`

Parameters: `c`: The color to set

textcolor

Description: Sets the current text color

Syntax: `void textcolor(unsigned char c)`

Parameters: `c`: The color to set

Notes: This function preserves attributes in the upper 4-bits if extended attributes are enabled. See setextendedattrib.

revers

Description: Enable the reverse attribute

Syntax: `void revers(unsigned char c)`

Parameters: `enable`: 0 to disable, 1 to enable

Notes: Extended attributes mode must be active. See setextendedattrib.

highlight

Description: Enable the highlight attribute

Syntax: `void highlight(unsigned char c)`

Parameters: `enable`: 0 to disable, 1 to enable

Notes: Extended attributes mode must be active. See setextendedattrib.

blink

Description: Enable the blink attribute

Syntax: `void blink(unsigned char c)`

Parameters: `enable`: 0 to disable, 1 to enable

Notes: Extended attributes mode must be active. See setextendedattrib.

underline

Description: Enable the underline attribute

Syntax: `void underline(unsigned char c)`

Parameters: `enable`: 0 to disable, 1 to enable

Notes: Extended attributes mode must be active. See setextendedattrib.

altpal

Description: Enable the alternate-palette attribute

Syntax: `void altpal(unsigned char c)`

Parameters: `enable`: 0 to disable, 1 to enable

Notes: Extended attributes mode must be active. See setextendedattrib.

clearattr

Description: Clear all text attributes

Syntax: `void clearattr()`

Notes: Extended attributes mode must be active. See setextendedattrib.

cellcolor

Description: Sets the color of a character cell

Syntax: `void cellcolor(unsigned char x, unsigned char y, unsigned char c)`

Parameters: **x:** The cell X-coordinate

y: The cell Y-coordinate

c: The color to set

Notes: No screen bounds checks are performed; out of screen behavior is undefined

setpalbank

Description: Set current text/bitmap palette bank (BTPALSEL).

Syntax: `void setpalbank(unsigned char bank)`

Parameters: **bank:** The palette bank to set. Valid values are 0, 1, 2 or 3.

Notes: Use setpalbanka to set alternate text/bitmap palette

setpalbanka

Description: Set alternate text/bitmap palette bank.

Syntax: `void setpalbanka(unsigned char bank)`

Parameters: **bank:** The palette bank to set. Valid values are 0, 1, 2 or 3.

Notes: Use setpalbank to set main text/bitmap palette

getpalbank

Description: Get selected text(bitmap) palette bank.

Syntax: `unsigned char getpalbank(void)`

Notes: Use getpalbanka to get alternate text(bitmap) selected palette

Return Value: The current selected main text(bitmap) palette bank.

getpalbanka

Description: Get selected alternate text(bitmap) palette bank.

Syntax: `unsigned char getpalbanka(void)`

Notes: Use getpalbank to get main text(bitmap) selected palette

Return Value: The current selected alternate text(bitmap) palette bank.

setmapedpal

Description: Set mape-in palette bank at \$D100-\$D3FF.

Syntax: `void setmapedpal(unsigned char bank)`

Parameters: `bank`: The palette bank to map-in. Valid values are 0, 1, 2 or 3.

getmapedpal

Description: Get mape-in palette bank at \$D100-\$D3FF.

Syntax: `unsigned char getmapedpal(void)`

setpalentry

Description: Set color entry for the mape-in palette

Syntax: `void setpalentry(unsigned char c, unsigned char r, unsigned char g, unsigned char b)`

Parameters: `c`: The palette entry index (0-255)

- r:** The red component value
- g:** The green component value
- b:** The blue component value

Notes: Use setmapedmal to bank-in the palette to modify

fillrect

Description: Fill a rectangular area with character and color value

Syntax: `void fillrect(const RECT *rc, unsigned char ch, unsigned char col)`

Parameters: **rc:** A RECT structure specifying the box coordinates

ch: A char code to fill the rectangle

col: The color to fill

Notes: No screen bounds checks are performed; out of screen behavior is undefined

box

Description: Draws a box with graphic characters

Syntax: `void box(const RECT *rc, unsigned char color, unsigned char style, unsigned char clear, unsigned char shadow)`

Parameters: **rc:** A RECT structure specifying the box coordinates

color: The color to use for the graphic characters

style: The style for the box borders. Can be set to
BOX_STYLE_NONE, BOX_STYLE_ROUNDED, BOX_STYLE_INNER,
BOX_STYLE_OUTER, BOX_STYLE_MID

clear: Set to 1 to clear the box interior with the selected color

shadow: Set to 1 to draw a drop shadow

Notes: No screen bounds checks are performed; out of screen behavior is undefined

hline

Description: Draws an horizontal line.

Syntax: `void hline(unsigned char x, unsigned char y, unsigned char len, unsigned char style)`

Parameters: **x:** The line start X-coordinate

y: The line start Y-coordinate

len: The line length

style: The style for the line. See HLINE_ constants for available styles.

Notes: No screen bounds checks are performed; out of screen behavior is undefined

vline

Description: Draws a vertical line.

Syntax: `void vline(unsigned char x, unsigned char y, unsigned char len, unsigned char style)`

Parameters: **x:** The line start X-coordinate

y: The line start Y-coordinate

len: The line length

style: The style for the line. See VLINE_ constants for available styles.

Notes: No screen bounds checks are performed; out of screen behavior is undefined

gohome

Description: Set the current position at home (0,0 coordinate)

Syntax: `void gohome(void)`

gotoxy

Description: Set the current position at X,Y coordinates

Syntax: `void gotoxy(unsigned char x, unsigned char y)`

Parameters: `x`: The new X-coordinate
`y`: The new Y-coordinate

Notes: No screen bounds checks are performed; out of screen behavior is undefined

gotox

Description: Set the current position X-coordinate

Syntax: `void gotox(unsigned char x)`

Parameters: `x`: The new X-coordinate

Notes: No screen bounds checks are performed; out of screen behavior is undefined

gotoy

Description: Set the current position Y-coordinate

Syntax: `void gotoy(unsigned char y)`

Parameters: `y`: The new Y-coordinate

Notes: No screen bounds checks are performed; out of screen behavior is undefined

moveup

Description: Move current position up

Syntax: `void moveup(unsigned char count)`

Parameters: `count`: The number of positions to move

Notes: No screen bounds checks are performed; out of screen behavior is undefined

movedown

Description: Move current position down

Syntax: `void Movedown(unsigned char count)`

Parameters: **count:** The number of positions to move

Notes: No screen bounds checks are performed; out of screen behavior is undefined

moveleft

Description: Move current position left

Syntax: `void Moveleft(unsigned char count)`

Parameters: **count:** The number of positions to move

Notes: No screen bounds checks are performed; out of screen behavior is undefined

moveright

Description: Move current position right

Syntax: `void Moveright(unsigned char count)`

Parameters: **count:** The number of positions to move

Notes: No screen bounds checks are performed; out of screen behavior is undefined

wherex

Description: Return the current position X coordinate

Syntax: `unsigned char wherex(void)`

Return Value: The current position X coordinate

wherey

Description: Return the current position Y coordinate

Syntax: `unsigned char wherey(void)`

Return Value: The current position Y coordinate

pcputc

Description: Output a single petscii character to screen at current position

Syntax: `void cputc(unsigned char c)`

Parameters: `c`: The petscii character to output

pcputsxy

Description: Output a petscii string at X,Y coordinates

Syntax: `void pcputsxy (unsigned char x, unsigned char y, const unsigned char* s)`

Parameters: `x`: The X coordinate where string will be printed

`y`: The Y coordinate where string will be printed

`s`: The petscii string to print

Notes: No pointer check is performed. If `s` is null or invalid, behavior is undefined

cputcxy

Description: Output a single petscii character at X,Y coordinates

Syntax: `void pcputcxy (unsigned char x, unsigned char y, unsigned char c)`

Parameters: `x`: The X coordinate where character will be printed

`y`: The Y coordinate where character will be printed

`c`: The petscii character to print

pcputs

Description: Output a petsci string at current position

Syntax: `void pcputs(const unsigned char* s)`

Parameters: `s`: The string to print

Notes: No pointer check is performed. If `s` is null or invalid, behavior is undefined

cputc

Description: Output a single screen code character to screen at current position

Syntax: `void cputc(unsigned char c)`

Parameters: `c`: The screen code of the character to output

cputnc

Description: Output N copies of a character at current position

Syntax: `void cputnc(unsigned char count, unsigned char c)`

Parameters: `c`: The screen code of the characters to output

`count`: The count of characters to print

cputhex

Description: Output an hex-formatted number at current position

Syntax: `void cputhex(long n, unsigned char prec)`

Parameters: `n`: The number to write

`prec`: The precision of the hex number, in digits. Leading zeros will be printed accordingly

Notes: The \$ symbol will be automatically added at beginning of string

cputdec

Description: Output a decimal number at current position

Syntax: `void cputdec(long n, unsigned char padding, unsigned char leadingZ)`

Parameters: **n:** The number to write

padding: The padding space to add before number

leadingZ: The leading zeros to print

cputs

Description: Output screen codes at current position

Syntax: `void cputs(const unsigned char* s)`

Parameters: **s:** An array of screen codes to print

Notes: This function works with screen codes only. To output ordinary ASCII/PETSCII strings, use the "pcputs" macro. No pointer check is performed. If s is null or invalid, behavior is undefined.

cputsxy

Description: Output multiple screen codes at X,Y coordinates

Syntax: `void cputsxy (unsigned char x, unsigned char y, const unsigned char* s)`

Parameters: **x:** The X coordinate where string will be printed

y: The Y coordinate where string will be printed

s: An array of screen codes to print

Notes: This function works with screen codes only. To output ordinary ASCII/PETSCII strings, use the "pcputsxy" macro. No pointer check is performed. If s is null or invalid, behavior is undefined.

cputcxy

Description: Output a single character at X,Y coordinates

Syntax: `void cputcxy (unsigned char x, unsigned char y, unsigned char c)`

Parameters: **x:** The X coordinate where character will be printed
y: The Y coordinate where character will be printed
c: The screen code of the character to print

cputncxy

Description: Output N copies of a single character at X,Y coordinates

Syntax: `void cputncxy (unsigned char x, unsigned char y, unsigned char count, unsigned char c)`

Parameters: **x:** The X coordinate where character will be printed
y: The Y coordinate where character will be printed
count: The number of characters to output
c: The screen code of the characters to print

cprintf

Description: Prints formatted output.

Escape strings can be used to modify attributes, move cursor, etc similar to PRINT in CBM BASIC.

Syntax: `unsigned char cprintf (const unsigned char* format, ...)`

Parameters: **format:** The string to output. The available escape codes are:

Cursor positioning

\t Go to next tab position (multiple of 8s)

\r Carriage Return

\n New line

{clr} Clear screen {home} Move cursor to home (top-left)

{d} Move cursor down {u} Move cursor up

{r} Move cursor right {l} Move cursor left

Attributes

{rvson} Reverse attribute ON {rvsoff} Reverse attribute OFF
{blon} Blink attribute ON {bloff} Blink attribute OFF
{ulon} Underline attribute ON {uloff} Underline attribute OFF

Colors (default palette)

{blk} {wht} {red} {cyan}
{pur} {grn} {blu} {yel}
{ora} {brn} {pink} {gray1}
{gray2} {lblu} {lgrn} {gray3}

Notes:

This function works with screen codes only! To output ordinary ASCII/PETSCII strings, use the "pcprintf" macro. Currently no argument replacement is done with the variable arguments.

pcprintf

Description: Prints formatted petscii string output.

Syntax: `see cprintf`

cgetc

Description: Waits until a character is in the keyboard buffer and returns it

Syntax: `unsigned char cgetc (void);`

Return Value: The last character in the keyboard buffer

Notes: Returned values are ASCII character codes

kbhit

Description: Returns the character in the keyboard buffer

Syntax: `unsigned char kbhit (void);`

Return Value: The character code in the keyboard buffer, 0 otherwise.

Notes: Returned values are ASCII character codes

getkeymodstate

Description: Return the key modifiers state.

Syntax: `unsigned char getkeymodstate(void)`

Return Value: A byte with the key modifier state bits, where bits:

Bit	Meaning	Constant
0	Right SHIFT State	KEYMOD_RSHIFT
1	Left SHIFT state	KEYMOD_LSHIFT
2	CTRL state	KEYMOD_CTRL
3	MEGA state	KEYMOD_MEGA
4	ALT state	KEYMOD_ALT
5	NOSCRL state	KEYMOD_NOSCRL
6	CAPSLOCK state	KEYMOD_CAPSLOCK
7	Reserved	-

flushkeybuf

Description: Flush the keyboard buffer

Syntax: `void flushkeybuf(void)`

cinput

Description: Get input from keyboard, printing incoming characters at current position.

Syntax: `unsigned char cinput(char* buffer, unsigned char buflen, unsigned char flags)`

Parameters: **buffer:** Target character buffer preallocated by caller

buflen: Target buffer length in characters, including the null character terminator

flags: Flags for input: (default is accept all printable characters)

`CINPUT_ACCEPT_NUMERIC`
Accepts numeric characters.

`CINPUT_ACCEPT_LETTER`
Accepts letters.

CINPUT_ACCEPT_SYM

Accepts symbols.

CINPUT_ACCEPT_ALL

Accepts all. Equals to CINPUT_ACCEPT_NUMERIC
|CINPUT_ACCEPT LETTER |CINPUT_ACCEPT_SYM

CINPUT_ACCEPT_ALPHA

Accepts alphanumeric characters. Equals to
CINPUT_ACCEPT_NUMERIC |CINPUT_ACCEPT LETTER

CINPUT_NO_AUTOTRANSLATE

Disables the feature that makes cinput to autodisplay uppercase characters when standard lowercase character set is selected and the user enters letters without the SHIFT key, that would display graphic characters instead of alphabetic ones.

Return Value: Count of successfully read characters in buffer

VIC_BASE

VIC_BASE is a pre-processor macro that provides the base address of the VIC-IV chip, i.e., \$D000.

IS_H640 is a pre-processor macro that returns 0 if the current VIC-III/IV video mode is set to 320 pixels accross (40 column mode), and non-zero if it is set to 640 pixels across (80 column mode).

18

CHAPTER

BASIC Tokenisers

Various tokenisers for C64 BASIC exist, e.g., <https://github.com/catseye/hatoucan>, <https://www.c64-wiki.com/wiki/C64list>, or the **petcat** utility that is part of VICE. If you are using Ubuntu Linux, you can install **petcat** by using the following command:

```
sudo apt-get install vice
```

We recommend **petcat**, because it supports both C64 BASIC 2 and C65 BASIC 10. Some IDEs offer BASIC 65 tokenisers within them, such as:

Eleven

<https://files.mega65.org?id=8b189d0b-ea1e-45a7-a4de-87bcb0b11696>

C64 Studio

<https://www.georg-rottensteiner.de/files/C64StudioRelease.zip>

CBM prg Studio

<https://www.ajordison.co.uk>

PART VII

APPENDICES

APPENDIX A

Accessories

B

APPENDIX

BASIC 65 Command Reference

- **Commands, Functions and Operators**
- **BASIC Command Reference**

COMMANDS, FUNCTIONS AND OPERATORS

This appendix describes each of the commands, functions, and other callable elements of BASIC 65, which is an enhanced version of BASIC 10. Some of these can take one or more arguments, which are pieces of input that you provide as part of the command or function call, to help describe what you want to achieve. Some also require that you use special words.

Below is an example of how commands, functions, and operators (all of which are also known as keywords) will be described in this appendix.

KEY number, string

Here, **KEY** is a keyword. Keywords are special words that BASIC understands. Keywords are always written in **BOLD CAPITALS**, so that you can easily recognise them.

The words not in bold must be replaced for the command, function or operator to work. In this example, we need to replace number with a numeric expression, and string with a string expression. We'll explain what expressions are a bit more in a few moments.

The comma, and some other symbols and punctuation marks, just represent themselves when they are in bold. In our example here, it means that there must be a comma between the number and the string.

You might also see symbols and punctuation marks that are not in bold. When they are not in bold they have a special meaning. You might 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 the command, function or operator will also 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.

This arrangement of keywords, expressions and symbols is what's called syntax. If you miss something out, or put the wrong thing in the wrong place, it is called a syntax error. The computer will tell you that you have a syntax error by displaying a **?SYNTAX ERROR** message.

There is nothing to worry about if you get an error from the MEGA65. Instead, it is just the MEGA65's way of telling you that something isn't quite right, so that you can more easily find and fix the problem. Error messages such as this won't hurt the computer or cause any damage to 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 MEGA65 will respond with a **?SYNTAX ERROR**, similar to what's shown below:

```
KEY 8"FISH"
```

```
?SYNTAX ERROR
```

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

```
?SYNTAX ERROR
```

It is very common for commands, functions and operators to use one or more “**expressions**”. An expression is just a fancy name for something that has, or equates to a value. This could be as simple as a string (“HELLO”), a number (23.7), or a complex calculation that might include one or more functions or operators (`LEN("HELLO") * (3 XOR 7)`). Generally speaking, expressions can result in either a string or a numeric result. In this case we call the expressions 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 MEGA65 will display a **?TYPE MISMATCH ERROR**, to say that the type of expression you gave doesn’t match what it expected. 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"
```

If you wish, you can try typing this in yourself.

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
```

You can place a sequence of statements within a single line by separating them with colons, for example:

```
PRINT "HELLO" : PRINT "HOW ARE YOU?" : PRINT "HOW IS THE WEATHER?"
HELLO
HOW ARE YOU?
HOW IS THE WEATHER?
```

Direct Mode Commands

Note that some commands are said to only work in **direct mode**. This means that the command can't be part of a BASIC program, but can be entered directly to the screen. In the two **PRINT** examples above, the first was entered in direct mode, whereas the second one wasn't. The examples above would work since **PRINT** works in both direct and indirect mode.

Command Format Syntax

The following table describes what the other symbols found in this appendix mean.

Symbol	Meaning
...	The bracket can be repeated zero or more times
[]	Optional
< >	Include one of the choices
[]	Optionally include one of the choices
{ , }	One or more of the arguments is required. The commas to the left of the last argument included are required. Trailing commas must be omitted. See CURSOR for an example.
[{ , }]	Similar to { , } but all arguments can be omitted

Fonts

Whenever there's a piece of text in this appendix that reflects some logic, something you can type, or something the MEGA65 could display, the text will **WILL USE THIS FONT**. This helps make it easier for you to distinguish between these things and the written text.

BASIC 65 Constants

Type	Example	Example
Decimal Integer	32000	-55
Decimal Fixed Point	3.14	-7654.321
Decimal Floating Point	1.5E03	7.7E-02
Hex	\$0020	\$FF
String	"X"	"TEXT"

BASIC 65 Variables

Each scalar variable consumes 8 bytes of storage in memory. The reserved area in bank 0 from \$F700-\$FEFF can store 256 variables. Variables don't need to be declared, and their type is determined by an appended character. All variables without an appended character are regarded as REAL by default, and storage is claimed at their first usage. They are also initialised to zero, whereas string variables are initialised as an empty string "".

All 104 one-letter variables are declared as **fast** variables. 26 user functions are declared as **fast** functions. These are the variables (A - Z), (A% - Z%), (A& - Z&) and (A\$ - Z\$) and the functions (FNA() - FNZ()). They have fixed memory addresses in the range \$FD00 - \$FEFF, the address is generated by a hash algorithm from the variable name. The access to these variables and functions, either use or definition, is much faster, than the access to two letter variables and functions. The address of fast variables and functions is computed by a very fast algorithm, while the address of two-letter variables and functions is stored in a table, which has to be searched for every use.

Type	Appended Character	Range	Example
Byte	&	0 .. 255	BY& = 23
Integer	%	-32768 .. 32767	I% = 5
Real	none	-1E37 .. 1E37	XY = 1/3
String	\$	length = 0 .. 255	AB\$ = "TEXT"

BASIC 65 Arrays

Each array consumes the number of elements multiplied by the item size, plus the size of the header (6 + 2 * dimensions) in memory. For example the array

```
100 DIM X(8,2,3)
```

has 3 dimensions and 108 ($9 \times 3 \times 4$) items. You might be asking: Why is it $9 \times 3 \times 4$, when the program uses 8, 2, and 3? This is because array indexes start at 0, not 1.

The size for real items is 5, so the data of that array above would occupy 540 (5×108) bytes. The header size is 12 bytes ($6 + 2 * 3$), so the total length in memory is 552 bytes ($540 + 12$).

Arrays are stored in bank 1 starting at address \$2000 and expand upwards. They share the available memory at \$2000 .. \$F6FF with the string area, which starts in bank 1 at address \$F6FF, and expands downwards. Each of the above scalar variable types can be used as an array, by declaring them with a **DIM** statement. The arrays are initialised to zero for all elements on declaration. If an undeclared array element is used, an automatic implicit declaration is performed, which sets the upper boundary for each dimension to 10. For example, the usage of an undeclared element **AB(3,5)** would automatically perform a **DIM AB(10,10)**. As noted previously, the lower boundary for each dimension is always 0 (zero), so an array initialised with **DIM AB(10)** consists of 11 elements and accepts indexes from 0 to 10.

String arrays are more precisely expressed as *arrays of string descriptors*. Each item consists of three bytes, which hold these values: The length of the string, and the 2 byte address (low/high byte) of the string in string memory. The usage of the BASIC function **POINTER** with a string or string array element as the argument, returns the address of the descriptor, not the string itself.

Type & Item Size	Appended Character	Range	Example
Byte Array	1	&	<code>BV&(5,6) = 23</code>
Integer Array	2	%	<code>I%<0,10> = 5</code>
Real Array	5	none	<code>X%<I%> = 1/3</code>
String Array	3	\$	<code>AB\$(X) = "TEXT"</code>

Screen memory and colour RAM

Many programmers like to skip the standard routine **PRINT** for screen output and want to access the screen memory and the colour RAM directly using the commands **PEEK** and **POKE**. **BASIC 65** provides a method for direct access, that is easier to use and faster than **POKE**:

Two byte arrays are predefined, that are mapped to screen text memory and screen colour memory.

TEK<(COLUMNS,ROWS> is mapped to the text screen memory.

CCE<(COLUMNS,ROWS> is mapped to the colour screen memory.

In 80 column mode the arrays are dimensioned as:

T0&(79,24) Column = 0 - 79, Row = 0 - 24.

C0&(79,24) Column = 0 - 79, Row = 0 - 24.

In 40 column mode the arrays are dimensioned as:

T0&(39,24) Column = 0 - 39, Row = 0 - 24.

C0&(39,24) Column = 0 - 39, Row = 0 - 24.

If you want to display the character **A** (display code = 1) in row 2 and column 5 and make it yellow, you can either use the POKE method:

POKE 2213,1 address = $2048 + 2 * 80 + 5 = 2213$

POKE \$FF808A\$5,7 7 = YELLOW.

or use the mapped byte arrays:

T0&(5,2) = 1 character A

C0&(5,2) = 7 colour YELLOW

Examining the contents of a screen position is equally easy. To get the character and the colour at the left most (home) position:

T% = T0&(0,0)

C% = C0&(0,0)

T% will then contain the display code of the character, while C% has the colour index.

BASIC 65 Operators

BASIC 65 provides a set of operators that are typical of most BASIC programming dialects. The usage and precedence of these operators is documented in this section.

The = symbol is used both as an assignment operator, and as a relational operator for testing equality. For example, in the statement **A = B = 5**, the first equal sign is the assignment operator, while the second is a logical operator, comparing the variable **B** with 5. The value of **A** will either be assigned the value -1 (for **TRUE**), or the value 0 (for **FALSE**). You may have noticed that the value of -1 for **TRUE** is different to other programming languages, such as **C**, where the value of 1 is used for **TRUE** instead.

The + symbol can be used as a positive sign for numerical expressions, as an addition operator, or for string concatenation. The number and type of operands determines the operation.

The - symbol can be used as a negative sign for numerical expressions, or as a subtraction operator. The number and type of operands determines the operation.

The operators **NOT**, **AND**, **OR** and **XOR** can be used both as logical operators, or as boolean operators.

- Logical Operator Example: IF A>B AND A<0
- Boolean Operator Example: A = B AND \$7F

Both examples always produce an integer result internally, which can be interpreted either numerically or logically. If the result of a comparison is **TRUE**, the value will be set to **-1**, while a **FALSE** result yields **0**. In the boolean operator example above, the **AND** operator converts both operands to a 16-bit integer value, and performs a bitwise **AND** for all 16 bits. This example will take the value of **B**, set the upper 9 bits to zero, and store the result in **A**.

The result of logical operations can be used in numerical expressions as well, for example, **A = A - (B > ?)** will increment **A** by 1 if the result of **(B > ?)** is **TRUE** (-1). This is because the mathematical expression of **A = A - (-1)** is the same as **A = A + 1**.

The operators have precedences, which are listed in the tables below. In the statement **A * A - B * B** both multiplications will be performed first, before the subtraction is executed. Parentheses are used to change the precedence, for example **A * (A - B) * B** will execute the subtraction first.

Assignment Operator

Symbol	Description	Examples
=	Assignment	A = 42, A\$ = "HELLO", A = B < 42

Unary Mathematical Operators

Name	Symbol	Description	Example
Plus	+	Positive sign	A = +42
Minus	-	Negative sign	B = -42

Binary Mathematical Operators

Name	Symbol	Description	Example
Plus	+	Addition	$A = B + 42$
Minus	-	Subtraction	$B = A - 42$
Asterisk	*	Multiplication	$C = A * B$
Slash	/	Division	$D = B / 13$
Up Arrow	\uparrow	Exponentiation	$E = 2 \uparrow 10$
Left Shift	\ll	Left Shift	$A = B \ll 2$
Right Shift	\gg	Right Shift	$A = B \gg 1$

Note that the \uparrow character used for exponentiation is entered with  , which is next to 

Relational Operators

Symbol	Description	Example
$>$	Greater Than	$A > 42$
\geq	Greater Than or Equal To	$B \geq 42$
$<$	Less Than	$A < 42$
\leq	Less Than or Equal To	$B \leq 42$
$=$	Equal	$A = 42$
\neq	Not Equal	$B \neq 42$

Logical Operators

Keyword	Description	Example
AND	And	$A > 42 \text{ AND } A < 84$
OR	Or	$A > 42 \text{ OR } A = 0$
XOR	Exclusive Or	$A > 42 \text{ XOR } B > 42$
NOT	Negation	$C = \text{NOT } A > B$

Boolean Operators

Keyword	Description	Example
AND	And	A = B AND \$FF
OR	Or	A = B OR \$80
XOR	Exclusive Or	A = B XOR 1
NOT	Negation	A = NOT 22

String Operator

Name	Symbol	Description	Operand type	Example
Plus	+	Concatenates Strings	String	A\$ = B\$ + ".PRG"

Operator Precedence

Precedence	Operators
High	↑ + - (Unary Mathematical) * / + - (Binary Mathematical) << >> (Arithmetic Shifts) (<=) := () NOT
Low	AND OR XOR

Keywords And Tokens Part 1

*	AC	COLLECT	F3	EXP	BD
+	AA	COLLISION	FE17	FAST	FE25
-	AB	COLOR	E7	FGOSUB	FE48
/	AD	CONCAT	FE13	FGOTO	FE47
<	B3	CONT	9A	FILTER	FE03
<<	FE52	COPY	F4	FIND	FE2B
=	B2	COS	BE	FN	A5
>	B1	CURSOR	FE41	FONT	FE46
>>	FE53	CUT	E4	FOR	81
ABS	B6	DATA	83	BACKGROUND	FE39
AND	AF	DCLEAR	FE15	FORMAT	FE37
APPEND	FEOE	DCLOSE	FEOF	FRE	B8
ASC	C6	DEC	D1	FREAD#	FE1C
ATN	C1	DEF	96	FREEZER	FE4A
AUTO	DC	DELETE	F7	FWRITE#	FE1E
BACKGROUND	FE3B	DIM	86	GCOPY	FE32
BACKUP	F6	DIR	EE	GET	A1
BANK	FE02	DISK	FE40	GO	CB
BEGIN	FE18	DLOAD	F0	GOSUB	8D
BEND	FE19	DMA	FE1F	GOTO	89
BIT	FE4E	DMODE	FE35	GRAPHIC	DE
BLOAD	FE11	DO	EB	HEADER	F1
BOOT	FE1B	DOPEN	FE0D	HELP	EA
BORDER	FE3C	DOT	FE4C	HEX\$	D2
BOX	E1	DPAT	FE36	HIGHLIGHT	FE3D
BSAVE	FE10	DSAVE	EF	IF	8B
BUMP	CE03	DVERIFY	FE14	INFO	FE4D
BVERIFY	FE28	ECTORY	FE29	INPUT	85
CATALOG	FE0C	EDIT	FE45	INPUT#	84
CHANGE	FE2C	EDMA	FE21	INSTR	D4
CHAR	E0	ELLIPSE	FE30	INT	B5
CHDIR	FE4B	ELSE	D5	JOY	CF
CHR\$	C7	END	80	KEY	F9
CIRCLE	E2	ENVELOPE	FEOA	LEFT\$	C8
CLOSE	A0	ERASE	FE2A	LEN	C3
CLR	9C	ERR\$	D3	LET	88
CMD	9D	EXIT	ED	LINE	E5

Keywords And Tokens Part 2

LIST	9B	PRINT	99	SOUND	DA
LOAD	93	PRINT#	98	SPC(A6
LOADIFF	FE43	PUDEF	DD	SPEED	FE26
LOCK	FE50	RCOLOR	CD	SPRCOLOR	FE08
LOG	BC	RCURSOR	FE42	SPRITE	FE07
LOG10	CE08	READ	87	SPRSAV	FE16
LOOP	EC	RECORD	FE12	SQR	BA
LPEN	CE04	REM	8F	STEP	A9
MEM	FE23	RENAME	F5	STOP	90
MERGE	E6	RENUMBER	F8	STR\$	C4
MID\$	CA	RESTORE	8C	SYS	9E
MKDIR	FE51	RESUME	D6	TAB(A3
MOD	CEO8	RETURN	8E	TAN	CO
MONITOR	FA	RGRAPHIC	CC	TEMPO	FE05
MOUNT	FE49	RIGHT\$	C9	THEN	A7
MOUSE	FE3E	RMOUSE	FE3F	TO	A4
MOVSPR	FE06	RND	BB	TRAP	D7
NEW	A2	RPALETTE	CE0D	TROFF	D9
NEXT	82	RPEN	DO	TRON	D8
NOT	A8	RPLAY	CE0F	TYPE	FE27
OFF	FE24	RREG	FE09	UNLOCK	FE4F
ON	91	RSPCOLOR	CE07	UNTIL	FC
OPEN	9F	RSPEED	CE0E	USING	FB
OR	B0	RSPPOS	CE05	USR	B7
PAINT	DF	RSprite	CE06	VAL	C5
PALETTE	FE34	RUN	8A	VERIFY	95
PASTE	E3	RWINDOW	CE09	VIEWPORT	FE31
PEEK	C2	SAVE	94	VOL	DB
PEN	FE33	SAVEIFF	FE44	VSYNC	FE54
PIXEL	CEO8	SCNCLR	E8	WAIT	92
PLAY	FE04	SCRATCH	F2	WHILE	FD
POINTER	CEO8	SCREEN	FE2E	WINDOW	FE1A
POKE	97	SET	FE2D	WPEEK	CE10
POLYGON	FE2F	SGN	B4	WPOKE	FE1D
POS	B9	SIN	BF	XOR	E9
POT	CE02	SLEEP	FE0B	~	AE

Tokens And Keywords Part 1

80 END	A5 FN	CA MID\$
81 FOR	A6 SPC(CB GO
82 NEXT	A7 THEN	CC RGRAPHIC
83 DATA	A8 NOT	CD RCOLOR
84 INPUT#	A9 STEP	CF JOY
85 INPUT	AA +	DO RPEN
86 DIM	AB -	D1 DEC
87 READ	AC *	D2 HEX\$
88 LET	AD /	D3 ERR\$
89 GOTO	AE ^	D4 INSTR
8A RUN	AF AND	D5 ELSE
8B IF	BO OR	D6 RESUME
8C RESTORE	B1 >	D7 TRAP
8D GOSUB	B2 =	D8 TRON
8E RETURN	B3 <	D9 TROFF
8F REM	B4 SGN	DA SOUND
90 STOP	B5 INT	DB VOL
91 ON	B6 ABS	DC AUTO
92 WAIT	B7 USR	DD PUDEF
93 LOAD	B8 FRE	DE GRAPHIC
94 SAVE	B9 POS	DF PAINT
95 VERIFY	BA SQR	E0 CHAR
96 DEF	BB RND	E1 BOX
97 POKE	BC LOG	E2 CIRCLE
98 PRINT#	BD EXP	E3 PASTE
99 PRINT	BE COS	E4 CUT
9A CONT	BF SIN	E5 LINE
9B LIST	CO TAN	E6 MERGE
9C CLR	C1 ATN	E7 COLOR
9D CMD	C2 PEEK	E8 SCNCLR
9E SYS	C3 LEN	E9 XOR
9F OPEN	C4 STR\$	EA HELP
A0 CLOSE	C5 VAL	EB DO
A1 GET	C6 ASC	EC LOOP
A2 NEW	C7 CHR\$	ED EXIT
A3 TAB(C8 LEFT\$	EE DIR
A4 TO	C9 RIGHT\$	EF DSAVE

Tokens And Keywords Part 2

F0 DLOAD	FE09 RREG	FE2F POLYGON
F1 HEADER	FE0A ENVELOPE	FE30 ELLIPSE
F2 SCRATCH	FE0B SLEEP	FE31 VIEWPORT
F3 COLLECT	FE0C CATALOG	FE32 GCOPY
F4 COPY	FE0D DOPEN	FE33 PEN
F5 RENAME	FE0E APPEND	FE34 PALETTE
F6 BACKUP	FE0F DCLOSE	FE35 DMODE
F7 DELETE	FE10 BSAVE	FE36 DPAT
F8 RENUMBER	FE11 BLOAD	FE37 FORMAT
F9 KEY	FE12 RECORD	FE39 FOREGROUND
FA MONITOR	FE13 CONCAT	FE3B BACKGROUND
FB USING	FE14 DVERIFY	FE3C BORDER
FC UNTIL	FE15 DCLEAR	FE3D HIGHLIGHT
FD WHILE	FE16 SPRSAV	FE3E MOUSE
CEO2 POT	FE17 COLLISION	FE3F RMOUSE
CEO3 BUMP	FE18 BEGIN	FE40 DISK
CEO4 LPEN	FE19 BEND	FE41 CURSOR
CEO5 RSPPPOS	FE1A WINDOW	FE42 RCURSOR
CEO6 RSPRITE	FE1B BOOT	FE43 LOADIFF
CEO7 RSPCOLOR	FE1C FREAD#	FE44 SAVEIFF
CEO8 LOG10	FE1D WPOKE	FE45 EDIT
CEO9 RWINDOW	FE1E FWRITE#	FE46 FONT
CEOA POINTER	FE1F DMA	FE47 FGOTO
CEOB MOD	FE21 EDMA	FE48 FGOSUB
CEOCC PIXEL	FE23 MEM	FE49 MOUNT
CEOED RPALETTE	FE24 OFF	FE4A FREEZER
CEOEE RSPEED	FE25 FAST	FE4B CHDIR
CEOOF RPLAY	FE26 SPEED	FE4C DOT
CE10 WPEEK	FE27 TYPE	FE4D INFO
FE02 BANK	FE28 BVERIFY	FE4E BIT
FE03 FILTER	FE29 ECTORY	FE4F UNLOCK
FE04 PLAY	FE2A ERASE	FE50 LOCK
FE05 TEMPO	FE2B FIND	FE51 MKDIR
FE06 MOVSPR	FE2C CHANGE	FE52 <<
FE07 SPRITE	FE2D SET	FE53 >>
FE08 SPRCOLOR	FE2E SCREEN	FE54 VSYNC

BASIC COMMAND REFERENCE

ABS

Token: \$B6

Format: **ABS(x)**

Usage: **ABS** returns the absolute value of the numeric argument **x**.

x numeric argument (integer or real expression).

Remarks: The result is of type real.

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: **AND** 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 (losing precision). Logical operands are converted to 16-bit integer using \$FFFF (decimal -1) for TRUE, and \$0000 (decimal 0) for FALSE.

Expression	Result
0 AND 0	0
0 AND 1	0
1 AND 0	0
1 AND 1	1

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

Examples: 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# channel, 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.

channel number, where:

- 1 <= **channel** <= 127 line terminator is CR.
- 128 <= **channel** <= 255 line terminator is CR LF.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

drive drive # in dual drive disk units.

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

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

Remarks: APPEND# works similarly to DOPEN#... ,W, except that the file must already exist. The content of the file is retained, and all printed text is appended to the end. Trying to APPEND to a non existing file reports a DOS error.

Examples: Open existing file in append mode:

```
APPEND#5,"DATA",U9  
APPEND#130,(DD$),U(UN%)  
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 was apparently chosen to be a mnemonic to ASCII, but the returned value is in fact the so-called PETSCII code.

Remarks: **ASC** returns zero for an empty string, whose behaviour is different to BASIC 2, where **ASC("")** gave an error. The inverse function to **ASC** is **CHR\$**. Refer to the **CHR\$** function on page [B-47](#) for more information.

Examples: Using **ASC**

```
PRINT ASC("MEGA")
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 $(-\pi/2$ 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**.

Examples: Using **ATN**

```
PRINT ATN(0.5)
.463647609
PRINT ATN(0.5) * 180 / pi
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 **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 disables it.

Examples: 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 of 0 to 255. All colours within this range are customisable via the **PALETTE** command. On startup, the MEGA65 only has the first 32 colours configured, which are described in the following table.

Colours: [Index and RGB values of colour palette](#)

Index	Red	Green	Blue	Colour
0	0	0	0	Black
1	15	15	15	White
2	15	0	0	Red
3	0	15	15	Cyan
4	15	0	15	Purple
5	0	15	0	Green
6	0	0	15	Blue
7	15	15	0	Yellow
8	15	6	0	Orange
9	10	4	0	Brown
10	15	7	7	Pink
11	5	5	5	Dark Grey
12	8	8	8	Medium Grey
13	9	15	9	Light Green
14	9	9	15	Light Blue
15	11	11	11	Light Grey
16	14	0	0	Guru Meditation
17	15	5	0	Rambutan
18	15	11	0	Carrot
19	14	14	0	Lemon Tart
20	7	15	0	Pandan
21	6	14	6	Seasick Green
22	0	14	3	Soylent Green
23	0	15	9	Slimer Green
24	0	13	13	The Other Cyan
25	0	9	15	Sea Sky
26	0	3	15	Smurf Blue
27	0	0	14	Screen of Death
28	7	0	15	Plum Sauce
29	12	0	15	Sour Grape
30	15	0	11	Bubblegum
31	15	3	6	Hot Tamales

Example: Using **BACKGROUND**

```
BACKGROUND 3 : REM SELECT BACKGROUND COLOUR CYAN
```

BACKUP

Token: \$F6

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

Usage: The first form of **BACKUP**, specifying units for source and target can only be used for the drives connected to the internal FDC (Floppy Disk Controller). Units 8 and 9 are reserved for this controller. These can be either the internal floppy drive (unit 8) and another floppy drive (unit 9) attached to the same ribbon cable, or mounted D81 disk images. Therefore, **BACKUP** can be used to copy from floppy to floppy, floppy to image, image to floppy and image to image, depending on image mounts and the existence of a second physical floppy drive.

The second form of **BACKUP**, specifying drives for source and target, is meant to be used for dual drive units connected to the IEC bus. For example: CBM 4040, 8050, 8250 via an IEEE-488 to IEC adapter. The backup is then done by the disk unit internally.

source unit or drive # of source disk.

target unit or drive # of target disk.

Remarks: The target disk will be formatted and an identical copy of the source disk will be written.

BACKUP cannot be used to backup from internal devices to IEC devices or vice versa.

Examples: Using **BACKUP**

```
BACKUP U8 TO U9      : REM BACKUP INTERNAL DRIVE 8 TO DRIVE 9
BACKUP U9 TO U8      : REM BACKUP DRIVE 9 TO INTERNAL DRIVE 8
BACKUP D0 TO D1, U10 : REM BACKUP ON DUAL DRIVE CONNECTED VIA IEC
```

BANK

Token: \$FE \$02

Format: **BANK** bank number

Usage: Selects the memory configuration for BASIC commands that use 16-bit addresses. These are **LOAD**, **LOADIFF**, **PEEK**, **POKE**, **SAVE**, **SYS**, and **WAIT**. Refer to the system memory map in Chapter/Appendix F on page F-3 for more information.

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, the BASIC and KERNEL ROM, 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: BEGIN and BEND act as a pair of braces around a compound statement to be executed after THEN or ELSE. 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, as 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$)>7 THEN 90  
50 BEND :REM IGNORE ALL EXCEPT (A-Z)  
60 IF A$<>CHR$(13) GOTO 10  
90 PRINT "PW=",PW$
```

BEND

Token: \$FE \$19

Format: BEGIN ... BEND

Usage: BEGIN and BEND act as a pair of braces around a compound statement to be executed after THEN or ELSE. 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 the 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] [,R] [,D drive] [,U unit]

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

BLOAD has two modes: The flat memory address mode can be used to load a program to any address in the 28-bit (256MB) address range where RAM is installed. This includes the standard RAM banks 0 to 5, as well as the 8MB of "attic RAM" at address \$8000000.

This mode is triggered by specifying an address at parameter P that is larger than \$FFFF. The bank parameter is ignored in this mode.

For compatibility reasons with older BASIC versions, **BLOAD** accepts the syntax with a 16-bit address at P and a bank number at B as well. The attic RAM is out of range for this compatibility mode.

The optional parameter **R** (RAW MODE) does not interpret or use the first two bytes of the program file as the load address, which is otherwise the default behaviour. In RAW MODE every byte is read as data.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

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 override 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 such as the 1541, 1571, or 1581.

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

Remarks: **BLOAD** cannot cross bank boundaries.

BLOAD uses the load address from the file, if no P parameter is given.

Examples: Using **BLOAD**

```
BLOAD "ML DATA", B0, U9
BLOAD "SPRITES"
BLOAD "ML ROUTINES", B1, P32768
BLOAD (FI$), B(BA%), P(PA%), U(UN%)
BLOAD "CHUNK",P($8000000)      :REM LOAD TO ATTIC RAM
```

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 in bank 0, and performs a **JSR \$0400** afterwards (Jump To Subroutine).

BOOT with no parameters attempts to load and execute a file named AUTOBOOT.C65 from the default unit 8. It's short for **RUN "AUTO-BOOT.C65"**.

filename is either a quoted string such as "**DATA**", or a string expression in brackets such as **(FI\$)**.

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 override 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 such as the 1541, 1571, or 1581.

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

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

Examples: Using **BOOT**

```
BOOT SYS
BOOT (FI$), 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 of 0 to 255. All colours within this range are customisable via the **PALETTE** command. On startup, the MEGA65 only has the first 32 colours configured, which are described in the table under **BACKGROUND** on page [B-25](#).

Example: Using **BORDER**

```
10 BORDER 4 : REM SELECT BORDER COLOUR PURPLE
```

BOX

Token: \$E1

Format: **BOX** x0,y0, x2,y2 [, solid]

BOX x0,y0, x1,y1, x2,y2, x3,y3 [, solid]

Usage: The first form of **BOX** with two coordinate pairs and an optional **solid** parameter draws a simple rectangle, assuming that the coordinate pairs declare two diagonally opposite corners.

The second form with four coordinate pairs declares a path of four points, which will be connected with lines. The path is closed by connecting the last coordinate with the first.

The quadrangle is drawn using the current drawing context set with **SCREEN**, **PALETTE** and **PEN**. The quadrangle is filled if the parameter **solid** is not 0.

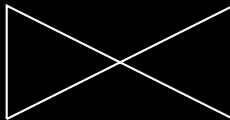
Remarks: **BOX** can be used with four coordinate pairs to draw any shape that can be defined with four points, not only rectangles. For example rhomboids, kites, trapezoids and parallelograms. It is also possible to draw bow tie shapes.

Examples: Using **BOX**

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



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



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



BSAVE

Token: \$FE \$10

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

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

BSAVE has two modes: The flat memory address mode can be used to save a memory block in the 28-bit (256MB) address range where RAM is installed. This includes the standard RAM banks 0 to 5, as well as the 8MB of "attic RAM" at address \$8000000.

This mode is triggered by specifying addresses for the start and end parameter **P**, that are larger than \$FFFF. The bank parameter is ignored in this mode. This flat memory mode allows saving ranges greater than 64K.

For compatibility reasons with older BASIC versions, **BSAVE** accepts the syntax with 16-bit addresses at **P** and a bank number at **B** as well. The attic RAM is out of range for this compatibility mode. This mode cannot cross bank boundaries, so start and end address must be in the same bank.

filename is either a quoted string such as "DATA", or a string expression in brackets such as **(F1\$)**. If the first character of the filename is an at sign '@', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

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

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

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

drive drive # in dual drive disk units.

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

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

Remarks: The length of the file is **end - start + 2**.
If the number after an argument letter is not a decimal number, it must be set in parenthesis, as shown in the third and fourth line of the examples.

The **PRG** file format that is used by **BSAVE** requires the load address to be written to the first two bytes. If the saving is done with a bank number that is not zero, or a start address greater than \$FFFF, this information will not fit. For compatibility reasons, only the the two low order bytes are written. Loading the file with the **BLOAD** command will then require the full 16-bit range of the load address as a parameter.

Examples: Using **BSAVE**

```
BSAVE "ML DATA", P 32768 TO P 33792, B0, U9  
BSAVE "SPRITES", P 1536 TO P 2058  
BSAVE "ML ROUTINES", B1, P($9000) TO P($A000)  
BSAVE (F1$), B(BX%), P(PA) TO P(PE), U(UN%)
```

BUMP

Token: \$CE \$03

Format: **BUMP**(type)

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

Remarks: It's possible to detect multiple collisions, but you will need to evaluate the sprite coordinates to detect which sprites have collided.

Example: Using **BUMP**

```
10 SX = BUMP(1) : REM SPRITE-SPRITE COLLISION
20 IF (SX AND 6) = 6 THEN PRINT "SPRITE 1 & 2 COLLISION"
30 REM ---
40 SX = BUMP(2) : REM SPRITE-DATA COLLISION
50 IF (SX <> 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 such as "DATA", or a string expression in brackets such as (FI\$).

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 that will be used.

drive drive # in dual drive disk units.

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

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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 **BVERIFY** 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.

Examples: Using **BVERIFY**

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

CATALOG

Token: \$FE \$0C

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

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

The **W** (Wide) parameter lists the directory three columns wide on the screen and pauses after the screen has been filled with a page (63 directory entries). Pressing any key displays the next page.

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

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

drive drive # in dual drive disk units.

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

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

Remarks: **CATALOG** is a synonym of **DIRECTORY** and **DIR**, and produces the same listing. The **filepattern** can be used to filter the listing. The wildcard characters * and ? may be used. Adding ,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.

The shortcut symbol **\$** can only be used in direct mode.

Examples: 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.
```

Below is an example showing how a directory looks with the **wide** parameter:

DIR W					
0 "BASIC EXAMPLES "					
1 "BEGIN"	P	1 "FREAD"	P	2 "PAINT.COR"	P
1 "BEND"	P	1 "FRE"	P	3 "PALETTE.COR"	P
1 "BUMP"	P	2 "GET#"	P	1 "PEEK"	P
1 "CHAR"	P	1 "GETKEY"	P	3 "PEN"	P
1 "CHRS"	P	1 "GET"	P	1 "PLAY"	P
4 "CIRCLE"	P	2 "GOSUB"	P	2 "POINTER"	P
1 "CLOSE"	P	2 "GOTO.COR"	P	1 "POKE"	P
1 "CLR"	P	2 "GRAPHIC"	P	1 "POS"	P
2 "COLLISION"	P	1 "HELP"	P	1 "POT"	P
1 "CURSOR"	P	1 "IF"	P	1 "PRINT#"	P
0 "DATA BASE"	R	2 "INPUT#"	P	1 "PRINT"	P
1 "DATA"	P	2 "INPUT"	P	1 "RCOLOR.COR"	P
1 "DEF FN"	P	2 "JOY"	P	1 "READ"	P
1 "DIM"	P	1 "LINE INPUT#"	P	1 "RECORD"	P
1 "DO"	P	3 "LINE"	P	1 "REM"	P
5 "ELLIPSE"	P	1 "LOOP"	P	1 "RESTORE"	P
1 "ELSE"	P	1 "MIDS"	P	1 "RESUME"	P
1 "EL"	P	1 "MOD"	P	1 "RETURN"	P
1 "ENVELOPE"	P	1 "MOUSPR"	P	1 "REVERS"	S
2 "EXIT"	P	1 "NEXT"	P	3 "RGRAPHIC"	P
1 "FOR"	P	2 "ON"	P	1 "RMOUSE"	P

CHANGE

Token: \$FE \$2C

Format: **CHANGE /findstring/ TO /replacestring/ [, line range]**

CHANGE "findstring" TO "replacestring" [, line range]

Usage: **CHANGE** performs a **find and replace** of the BASIC program that is currently in memory. An optional **line range** limits the search to this range, otherwise the entire BASIC program is searched. At each occurrence of the **findstring**, the line is listed and the user is prompted for an action:

- **Y** **RETURN** perform the replace and find the next string
- **N** **RETURN** do **not** perform the replace and find the next string
- ***** **RETURN** replace the current and all following matches
- **RETURN** exit the command, and don't replace the current match

Remarks: Any un-shifted character that is not part of the string can be used instead of **/**.

However, using the double quote character finds text strings that are not tokenised, and therefore not part of a keyword.

For example, **CHANGE "LOOP" TO "OOPS"** will not find the BASIC keyword **LOOP**, because the keyword is stored as a token and not as text. However **CHANGE /LOOP/ TO /OOPS/** will find and replace it (possibly causing **SYNTAX ERRORS**).

Can only be used in direct mode.

Examples: Using **CHANGE**

```
CHANGE "XX$" TO "UU$", 2000-2700
```

```
CHANGE /IN/ TO /OUT/
```

```
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 in all resolutions.

column (in units of character positions) is the start position of the output horizontally. As each column unit is 8 pixels wide, a screen width of 320 has a column range of 0-39, while a screen width of 640 has a column range of 0-79.

row (in pixel units) is the start position of the output vertically. In contrast to the column parameter, its unit is in pixels (not character positions), with the top row having the value of 0.

height is a factor applied to the vertical size of the characters, where 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, where 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 to the default character set at \$29800, which includes upper and lower case characters.

Three character sets (see also **FONT**) are available:

- **\$29000** Font A (ASCII)
- **\$3D000** Font B (Bold)
- **\$2D000** Font C (CBM)

The first part of the font (upper case / graphics) is stored at \$xx000 - \$xx7FF.

The second part of the font (lower case / upper case) is stored at \$xx800 - \$xxFFF.

string is a string constant or expression which will be printed. This string may optionally contain one or more of the following control characters:

Expression	Keyboard Shortcut	Description
CHR\$(2)	CTRL+B	Blank Cell
CHR\$(6)	CTRL+F	Flip Character
CHR\$(9)	CTRL+I	AND With Screen
CHR\$(15)	CTRL+O	OR With Screen
CHR\$(24)	CTRL+X	XOR With Screen
CHR\$(18)	RVSON	Reverse
CHR\$(146)	RVSOFF	Reverse Off
CHR\$(147)	CLR	Clear Viewport
CHR\$(21)	CTRL+U	Underline
CHR\$(25)+"-"	CTRL+Y + "-"	Rotate Left
CHR\$(25)+"+"	CTRL+Y + "+"	Rotate Right
CHR\$(26)	CTRL+Z	Mirror
CHR\$(157)	Cursor Left	Move Left
CHR\$(29)	Cursor Right	Move Right
CHR\$(145)	Cursor Up	Move Up
CHR\$(17)	Cursor Down	Move Down

Notice that the start position of the string has different units in the horizontal and vertical directions. Horizontal is in columns and vertical is in pixels.

Refer to the **CHR\$** function on page [B-47](#) for more information.

Remarks: Using **CHAR**

```
10 SCREEN 640,400,2
20 CHAR 28,180,4,4,2,"MEGA65",$29000
30 GETKEY A$
40 SCREEN CLOSE
```

Will print the text "MEGA65" at the centre of a 640 x 400 graphic screen.

CHARDEF

Token: \$E0 \$96

Format: **CHARDEF** index, bit-matrix

Usage: Change the bitmap matrix of characters

index is the character number in display code, (@:0, A:1, B:2, ...)

bit-matrix is a set of 8 byte values, which define the raster representation for the character from top row to bottom row. If more than 8 values are used as arguments, the values 9-16 are used for the character index+1, 17-24 for index+2, etc.

Remarks: The character bitmap changes are applied to the VIC character generator, which resides in RAM at the address \$FF7E000.

All changes are volatile and the VIC character set can be restored by a reset or by using the **FONT** command.

Examples: Using **CHARDEF**

```
CHARDEF 1,$FF,$81,$81,$81,$81,$81,$81,$FF :REM CHANGE 'A' TO RECTANGLE  
CHARDEF 9,$18,$18,$18,$18,$18,$18,$18,$00 :REM MAKE 'I' SANS SERIF
```

CHDIR

Token: \$FE \$4B

Format: **CHDIR** dirname [,U unit]

Usage: Change to a subdirectory or a parent directory.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (F1\$).

Dependent on the **unit**, **CHDIR** is applied to different filesystems.

UNIT 12 is reserved for the SD-Card (FAT filesystem). There this command can be used to navigate to subdirectories and mount disk images, that are stored there. **CHDIR "..",U12** changes to the parent directory on **UNIT 12**.

For units, that are managed by CBDOS (typically 8 and 9), **CHDIR** is used to change into or out of subdirectories on floppy or disk image of type **D81**. Existing subdirectories are displayed as filetype **CBM** in the parent directory, they are created with the command **MKDIR**. **CHDIR "/" ,U unit** changes to the root directory.

Examples: Using **CHDIR**

```
CHDIR "ADVENTURES",U12 :REM ENTER ADVENTURES ON SD CARD
```

```
CHDIR "..",U12 :REM GO BACK TO PARENT DIRECTORY
```

```
CHDIR "RACING",U12 :REM ENTER SUBDIRECTORY RACING
```

```
0 "MEGAB5" " 1D
```

```
800 "MEGAB5 GAMES" CBM
```

```
800 "MEGAB5 TOOLS" CBM
```

```
600 "BASIC PROGRAMS" CBM
```

```
960 BLOCKS FREE.
```

```
CHDIR "MEGAB5 GAMES",U8 :REM ENTER SUBDIRECTORY ON FLOPPY DISK
```

```
CHDIR "/",U8 :REM GO BACK TO ROOT DIRECTORY
```

CHR\$

Token: \$C1

Format: CHR\$(numeric expression)

Usage: Returns a string containing one character, whose PETSCII value is equal to the argument.

Remarks: The argument range is from 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. The complete table of characters (and their PETSCII codes) is on page [C-3](#).

Example: Using CHR\$

```
10 QUOTES = CHR$(34)
20 ESCAPE$ = CHR$(27)
30 PRINT QUOTES;"MEGA$5";QUOTE$ : REM PRINT "MEGA$5"
40 PRINT ESCAPE$;"Q";      : REM CLEAR TO END OF LINE
```

CIRCLE

Token: \$E2

Format: **CIRCLE** xc, yc, radius [, flags , start, stop]

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

xc is the x coordinate of the centre in pixels

yc is the y coordinate of the centre in pixels

radius is the radius of the circle in pixels

flags control filling, arcs and the position of the 0 degree angle. Default setting (zero) is don't fill, draw legs and the 0 degree radian points to 3 o' clock.

Bit	Name	Value	Action if set
0	fill	1	Fill circle or arc with the current pen colour
1	legs	2	Suppress drawing of the legs of an arc
2	combs	4	Let the zero radian point to 12 o' clock

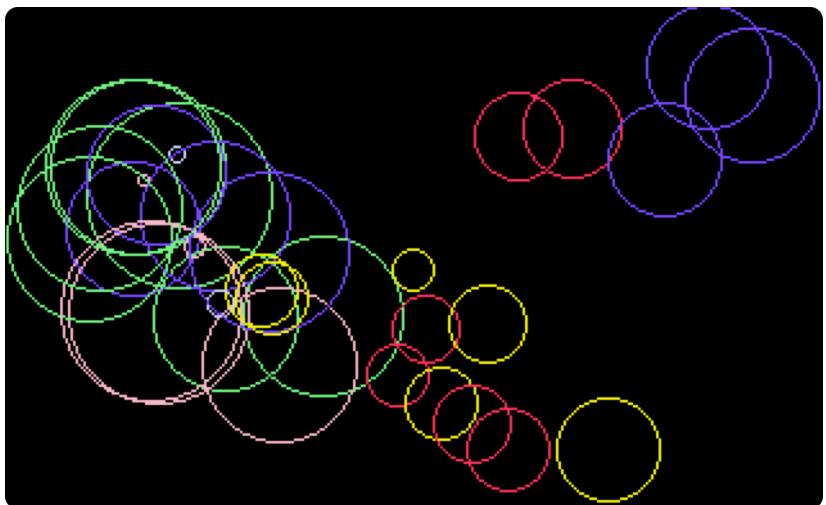
The units for the start- and stop-angle are degrees in the range of 0 to 360. The 0 radian starts at 3 o' clock and moves clockwise. Setting bit 2 of flags (value 4) moves the zero-radian to the 12 o' clock position.

start start angle for drawing an arc.

stop stop angle for drawing an arc.

Remarks: **CIRCLE** is used to draw circles on screens with an aspect ratio of 1:1 (for example: 320 x 200 or 640 x 400). Whilst using other resolutions (such as 640 x 200), the shape will be an ellipse instead.

The example program uses the random number function **RND** for circle colour, size and position. So it shows a different picture for each run.



Example: Using **CIRCLE**

```
100 REM CIRCLE (AFTER F.BOWEN)
110 BORDER 0 :REM BLACK
120 SCREEN 320,200,4 :REM SIMPLE SCREEN SETUP
130 PALETTE 0,0,0,0 :REM BLACK
140 PALETTE 0,1,RND(.)*16,RND(.)*16,15 :REM RANDOM COLOURS
150 PALETTE 0,2,RND(.)*16,15,RND(.)*16
160 PALETTE 0,3,15,RND(.)*16,RND(.)*16
170 PALETTE 0,4,RND(.)*16,RND(.)*16,15
180 PALETTE 0,5,RND(.)*16,15,RND(.)*16
190 PALETTE 0,6,15,RND(.)*16,RND(.)*16
200 SCNC LR 0 :REM CLEAR
210 FOR I=0 TO 32 :REM CIRCLE LOOP
220 PEN 0,RND(.)*6+1 :REM RANDOM PEN
230 R=RND(.)*36+1 :REM RADIUS
240 XC= R+RND(.)*320:IF(XC>319)THEN240:REM X CENTRE
250 YC= R+RND(.)*200:IF(YC>199)THEN250:REM Y CENTRE
260 XC=XC+WT*320:YC=YC+HT*200
270 CIRCLE XC,YC,R,. :REM DRAW
280 NEXT
290 GETKEY A$:REM WAIT FOR KEY
300 SCREEN CLOSE:BORDER 6
```

CLOSE

Token: \$A0

Format: **CLOSE** channel

Usage: Closes an input or output channel.

channel number, which was given to a previous call of commands such as **APPEND**, **DOPEN**, or **OPEN**.

Remarks: Closing files that have previously been opened before a program has completed is very important, especially for output files. **CLOSE** flushes output buffers and updates the directory information on disks. Failing to **CLOSE** can corrupt files and disks. BASIC does NOT automatically close channels nor files when a program stops.

Example: Using **CLOSE**

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

CLR

Token: \$9C

Format: **CLR**

CLR variable

Usage: Used for management of BASIC variables, arrays and strings. The run-time stack pointers, and the table of open channels is reset. After executing **CLR** all variables and arrays will be undeclared. **RUN** performs **CLR** automatically.

CLR variable clears (zeroes) the variable. **variable** can be a numeric variable or a string variable, but not an array.

Remarks: **CLR** should not be used inside loops or subroutines, as it destroys the return address. After **CLR**, all variables are unknown and will be initialised when they are next used.

Example: Using **CLR**

```
10 A=5: P$="MEGAB5"
20 CLR
30 PRINT A;P$
RUN

0
```

CLRBIT

Token: \$9C \$FE \$4E

Format: CLRBIT address, bit number

Usage: Clears (resets) a single bit at the **address**.

If the address is in the range of \$0000 to \$FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to \$10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

The **bit number** is a value in the range of 0-7.

A bank value > 127 is used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using CLRBIT

```
10 BANK 128      :REM SELECT SYSTEM MAPPING
20 CLRBIT $D011,4 :REM DISABLE DISPLAY
30 CLRBIT $D016,3 :REM SWITCH TO 38 OR 76 COLUMN MODE
```

CMD

Token: \$9D

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

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

channel number, which was given to a previous call of commands such as **APPEND**, **DOPEN**, or **OPEN**.

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

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

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

```
OPEN 1,4 :REM OPEN CHANNEL #1 TO PRINTER AT UNIT 4
CMD 1
LIST
PRINT#1
CLOSE 1
```

COLLECT

Token: \$F3

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

Usage: Rebuilds the **BAM** (Block Availability Map) of a disk, deleting splat files (files which have been opened, but not properly closed) 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 such as the 1541, 1571, or 1581.

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

Remarks: While this command is useful for cleaning a disk from splat files, 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 and may be overwritten by further disk write operations.

Examples: Using **COLLECT**

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

COLLISION

Token: \$FE \$17

Format: **COLLISION** type [, line number]

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

type specifies the collision type for this interrupt handler:

Type	Description
1	Sprite - Sprite Collision
2	Sprite - Data - Collision
3	Light Pen

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

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

Info: **COLLISION** wasn't completed in BASIC 10, and a working implementation will be available in a future BASIC 65 update.

Example: Using **COLLISION**

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

COLOR

Token: \$E7

Format: COLOR colour-index

Usage: The command works in the same way as **BACKGROUND**, i.e: sets the foreground colour (text colour) of the screen to the colour argument, which must be in the range of 0 to 31. Refer to the table under **BACKGROUND** on page [B-25](#) for the colour values and their corresponding colours.

Example: Using COLOR

```
10 COLOR ON : COLOR 2 : PRINT "COLOR ON AND RED"
20 COLOR OFF : COLOR 2 : PRINT "COLOR OFF (CAN'T CHANGE COLOUR)"
30 COLOR ON : COLOR 3 : PRINT "COLOR ON AND CYAN"

READY.

RUN
COLOR ON AND RED
COLOR OFF (CAN'T CHANGE COLOUR)
COLOR ON AND CYAN
```

CONCAT

Token: \$FE \$13

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

Usage: **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 brackets, for example: (FI\$)

targetfile is either a quoted string, for example: "SAFE" or a string expression in brackets, for example: (F\$S\$)

If the disk unit has dual drives, it is possible to apply **CONCAT** to files which are stored on different disks. In this case, it is necessary to specify the drive# for both files. This is also necessary 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 such as the 1541, 1571, or 1581.

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

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

Examples: 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 . This is a useful debugging tool. The BASIC program may be stopped and variables can be examined, and even changed. The **CONT** statement resumes execution.

Remarks: **CONT** cannot be used if a program has stopped because of an error. Also, any editing of a program inhibits continuation. Stopping and continuation can spoil the screen output, and can also 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] [,U unit] **TO** [target] [,D drive] [,U unit]

Usage: Copies the contents of **source** to **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 brackets, e.g. (F1\$).

target is either a quoted string, e.g. "BACKUP" or a string expression in brackets, e.g. (F\$).

drive drive # in dual drive disk units.

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

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

If none or one unit number is given, or the unit numbers before and after the TO token are equal, **COPY** is executed on the disk drive itself, and the source and target files will be on the same disk.

If the source unit (before TO) is different to the target unit (after TO), **COPY** is executed in MEGA65 BASIC by reading the source files into a RAM buffer and writing to the target unit. In this case, the target file name cannot be chosen, it will be the same as the source filename. The extended unit-to-unit copy mode allows the copying of single files, pattern matching files or all files of a disk. Any combination of units is allowed, internal floppy, D81 disk images, IEC floppy drives such as the 1541, 1571, 1581, or CMD floppy and hard drives.

Remarks: The file types **PRG**, **SEQ** and **USR** can be copied. If source and target are on the same disk, the target filename must be different from the source file name.

COPY cannot copy **DEL** files, which are commonly used as titles or separators in disk directories. These do not conform to Commodore DOS rules and cannot be accessed by standard **OPEN** routines.

REL files cannot be copied from unit to unit.

Examples: Using **COPY**

```
COPY U8 TO U9      :REM COPY ALL FILES  
COPY "CODES" TO "BACKUP" :REM COPY SINGLE FILE  
COPY "*.TXT",U8 TO U9   :REM PATTERN COPY  
COPY "M*U",U9 TO U11    :REM PATTERN COPY
```

COS

Token: \$BE

Format: **COS**(numeric expression)

Usage: 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 multiplying it with $\pi/180$.

Examples: Using **COS**

```
PRINT COS(0.7)
```

```
0.76484219
```

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

```
0.5
```

CURSOR

Format: **CURSOR <ON | OFF> [{, column, row, style}]**
 CURSOR {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 SCNCLR
20 CURSOR ON,1,2,1      :REM DISPLAY A SOLID CURSOR AT COLUMN 1, ROW 2
30 PRINT "A"; : SLEEP 1
40 CURSOR ,,0            :REM CHANGE TO A FLASHING CURSOR
50 PRINT "B"; : SLEEP 1
60 CURSOR OFF           :REM HIDE THE CURSOR
70 PRINT "C"; : SLEEP 1
80 CURSOR 20,10          :REM MOVE THE CURSOR TO COLUMN 20, ROW 10
90 PRINT "D"; : SLEEP 1
100 CURSOR ,50           :REM MOVE THE CURSOR TO ROW 5 BUT DO NOT CHANGE THE COLUMN
110 PRINT "E"; : SLEEP 1
100 CURSOR 0              :REM MOVE THE CURSOR TO THE START OF THE ROW
110 PRINT "F"; : SLEEP 1
```

CUT

Token: \$E4

Format: **CUT** x, y, width, height

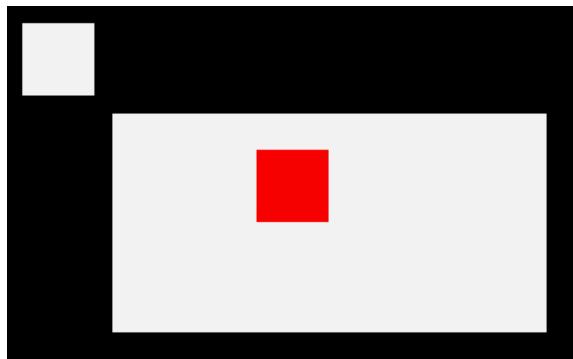
Usage: **CUT** is used on graphic screens and copies the content of the specified rectangle with upper left position **x, y** and the **width** and **height** to a buffer and fills the region afterwards with the colour of the currently selected pen.

The cut out can be inserted at any position with the command **PASTE**.

Remarks: The size of the rectangle is limited by the 1K size of the cut/copy/paste buffer. The memory requirement for a cut out region is width * height * number of bitplanes / 8. It must not equal or exceed 1024 byte. For a 4-bitplane screen for example, a 45 x 45 region needs 1012.5 byte.

Example: Using **CUT**

```
10 SCREEN 320,200,2
20 BOX 60,60,300,180,1 :REM DRAW A WHITE BOX
30 PEN 2 :REM SELECT RED PEN
40 CUT 140,80,40,40 :REM CUT OUT A 40 * 40 REGION
50 PASTE 10,10,40,40 :REM PASTE IT TO NEW POSITION
60 GETKEY A$ :REM WAIT FOR KEYPRESS
70 SCREEN CLOSE
```



DATA

Token: \$83

Format: **DATA** [constant [, constant ...]]

Usage: Used to define constants which can be read by **READ** statements in a program. Numbers and strings are allowed, but expressions are not. Items are separated by commas. Strings containing commas, colons or spaces must be placed in quotes.

RUN initialises the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is the programmer's responsibility 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.

RESTORE may be used to set the data pointer to a specific line for subsequent reads.

Remarks: It is good programming practice to put large amounts of **DATA** statements at the end of the program, so they don't slow down the search for line numbers after **GOTO**, and other statements with line number targets.

Example: Using **DATA**

```
1 REM DATA
10 READ N$, VE
20 READ NX : FOR I=2 TO NX : READ GL(I) : NEXT I
30 PRINT "PROGRAM:",N$," VERSION:",VE
40 PRINT "N-POINT GAUSSLEGENDRE FACTORS E1":
50 FOR I=2 TO NX:PRINT I,GL(I):NEXT I
60 END
80 DATA "MEGA65",1.1
90 DATA 5,0.5120,0.3573,0.2760,0.2252
```

```
RUN
PROGRAM:MEGA65 VERSION: 1.1
N-POINT GAUSSLEGENDRE FACTORS E1
2 0.512
3 0.3573
4 0.276
5 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 such as the 1541, 1571, or 1581.

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

The DOS of the disk drive will close all open files, clear all channels, free buffers and re-read the BAM. All open channels on the computer will also be closed.

Examples: Using **DCLEAR**

```
DCLEAR  
DCLEAR U9  
DCLEAR D0, U9
```

DCLOSE

Token: \$FE \$0F

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

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

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**.

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

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

Remarks: It is important to close all open files before a program ends. Otherwise buffers will not be freed and even worse, open files that have been written to may be incomplete (commonly called splat files), and no longer usable.

Examples: 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" (0 to 65535 in decimal). The argument must have 1-4 hex digits.

Remarks: Allowed digits in uppercase/graphics mode are 0-9 and A-Z (0123456789ABCDEF) and in lowercase/uppercase mode are 0-9 and a-z (0123456789abcdef).

Example: Using **DEC**

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

DEF FN

Token: \$96

Format: **DEF FN** name(real variable) = [expression]

Usage: Defines a single statement user function with one argument of type real, 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 when the function is used.

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

Example: Using **DEF FN**

```
10 PD = pi / 180
20 DEF FN CD(X)= COS(X*PD): REM COS FOR DEGREES
30 DEF FN SD(X)= SIN(X*PD): REM SIN FOR DEGREES
40 FOR D=0 TO 360 STEP 90
50 PRINT USING "###";D
60 PRINT USING " ##.##";FNC(D);
70 PRINT USING " ##.##";FNS(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 to either delete a range of lines from the BASIC program or to delete files from disk.

line range consists of the first and 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 brackets, for example: (F\$)

drive drive # in dual drive disk units.

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

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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: **DELETE filename** is a synonym of **SCRATCH filename** and **ERASE filename**.

Examples: 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
DELETE "*=SEQ"  :REM DELETE ALL SEQUENTIAL FILES
DELETE "R*=PRG" :REM DELETE PROGRAM FILES STARTING WITH 'R'
```

DIM

Token: \$86

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

Usage: Declares the shape, 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 as declared. The rules for variable names apply for array names as well. You can create byte arrays, 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: Byte arrays consume one byte per element, integer arrays two bytes, real arrays five bytes and string arrays three bytes for the string descriptor plus the length of the string itself.

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

Example: Using **DIM**

```
1 REM DIM
10 DIM A%(8) : REM ARRAY OF 9 ELEMENTS
20 DIM XX(2,3) : REM ARRAY OF 3X4 = 12 ELEMENTS
30 FOR I=0 TO 8 : A%(I)=PEEK(256+I) : PRINT A%(I);: NEXT:PRINT
40 FOR I=0 TO 2 : FOR J=0 TO 3 : READ XX(I,J):PRINT XX(I,J);: NEXT J,I
50 END
60 DATA 1,-2,3,-4,5,-6,7,-8,9,-10,11,-12

RUN
45 52 50 0 0 0 0 0 0
1 -2 3 -4 5 -6 7 -8 9 -10 11 -12
```

DIR

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

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

DIRECTORY [filepattern] [,W] [,R] [,D drive] [,U unit]

\$ [filepattern] [,W] [,R] [,D drive] [,U unit]

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

The **W** (Wide) parameter lists the directory three columns wide on the screen and pauses after the screen has been filled with a page (63 directory entries). Pressing any key displays the next page.

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: "M*" or a string expression in brackets, e.g. (DI\$)

drive drive # in dual drive disk units.

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

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

Remarks: **DIR** is a synonym of **CATALOG** and **DIRECTORY**, and produces the same listing. The **filepattern** can be used to filter the listing. The wildcard characters * and ? may be used. Adding ,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.

The shortcut symbol **\$** can only be used in direct mode.

Examples: Using **DIR**

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

For a **DIR** listing with the **wide** parameter, please refer to the example under **CATALOG** on page [B-41](#).

DISK

Token: \$FE \$40

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

 command [,U unit]

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

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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.

Using **DISK** with no parameters prints the disk status.

The shortcut key  can only be used in direct mode.

Examples: Using **DISK**

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

DLOAD

Token: \$F0

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

DLOAD "\$[pattern=type]" [,D drive] [,U unit]

DLOAD \$\$[pattern=type]" [,D drive] [,U unit]

Usage: The first form loads a file of type **PRG** into memory reserved for BASIC programs.

The second form loads a directory into memory, which can then be viewed with **LIST** or **LSTP**. It is structured like a BASIC program, but file sizes are displayed instead of line numbers.

The third form is similar to the second one, but the files are numbered. This listing can be scrolled like a BASIC program with the keys **F9** or **F11**, edited, listed, saved or printed.

A filter can be applied by specifying a pattern or a pattern and a type. The asterisk matches the rest of the name, while the ? matches any single character. The type specifier can be a character of (P,S,U,R), that is Program, Sequential, User, or Relative file.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

drive drive # in dual drive disk units.

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

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

Remarks: The load address, which is stored in the first two bytes of the file is ignored. The program is loaded into 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 be **RUN** or edited. It is possible to use **DLOAD** in a running program. This is called overlaying, or chaining. If you do this, then the newly 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 also be used by the newly loaded program.

Every **DLOAD** either program or directory, will replace contents (programs), that are currently in memory.

Examples: Using **DLOAD**

```
DLOAD "APOCALYPSE"  
DLOAD "MEGA TOOLS",U9  
DLOAD (FI$),U(UN%)  
  
DLOAD "$"           :REM LOAD WHOLE DIRECTORY - WITH FILE SIZES  
DLOAD "$$"          :REM LOAD WHOLE DIRECTORY - SCROLLABLE  
DLOAD "$$X*-P"      :REM DIRECTORY WITH PRG FILES STARTING with 'X'
```

DMA

Token: \$FE \$1F

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

Usage: **DMA** ("Direct Memory Access") is obsolete, and has been replaced by **EDMA**.

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: **DMA** has access to the lower 1MB address range organised in 16 banks of 64 K. To avoid this limitation, use **EDMA**, which has access to the full 256MB address range.

Examples: A sequence of **DMA** calls to demonstrate fast screen drawing operations

```
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, stencil, style, thick

Usage: "Display MODE" sets several parameters of the graphics context, which is used by drawing commands.

Mode	Values
jam	0 - 1
complement	0 - 1
stencil	0 - 1
style	0 - 3
thick	1 - 8

DO

Token: \$EB

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

DO [<**UNTIL** | **WHILE**> logical expression]
. . . statements [**EXIT**]
LOOP [<**UNTIL** | **WHILE**> logical expression]

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

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

Examples: 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#** channel, filename [,**L** [reclen]] [,**W**] [,**D** drive] [,**U** unit]

Usage: Opens a file for reading or writing.

channel number, where:

- **1 <= channel <= 127** line terminator is CR.
- **128 <= channel <= 255** line terminator is CR LF.

L indicates, that the file is a relative file, which is opened for read/write, as well as random access. The reclength is mandatory for creating relative files. For existing relative files, **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 such as "DATA", or a string expression in brackets such as (FI\$).

drive drive # in dual drive disk units.

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

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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 **PRG** files or ",U" for **USR** files.

If the first character of the filename is an at sign '@', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

Examples: 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
```

DOT

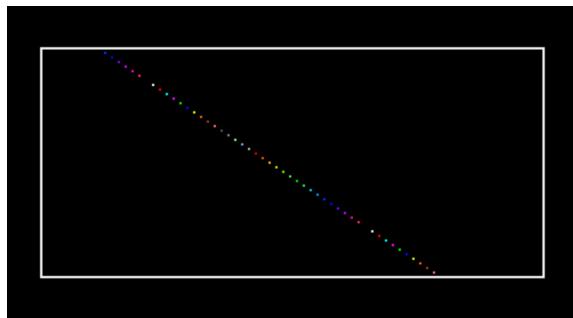
Token: \$FE \$4C

Format: **DOT** x, y [,colour]

Usage: Draws a pixel at screen coordinates x and y. The optional third parameter defines the colour to be used. If not specified, the current pen colour will be used.

Example: Using **DOT**:

```
10 SCREEN 320,200,5
20 BOX 50,50,270,150
30 VIEWPORT 50,50,220,100
40 FOR I=0 TO 127
50 DOT I+I+I,I+I,I
60 NEXT
70 GETKEY A
80 SCREEN CLOSE
```



DPAT

Token: \$FE \$36

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

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

There are four predefined pattern types, that can be selected by specifying the type number (1, 2, 3, or 4) as a single parameter.

A value of zero for the type number indicates a user defined pattern. This pattern can be set by using a bit string that consists of either 8, 16, 24, or 32 bits. The number of used pattern bytes is given as the second parameter. It defines how many pattern bytes (1, 2, 3, or 4) follow.

- **Type** 0-4
- **Number** number of following pattern bytes (1-4)
- **Pattern** pattern bytes

DS

Format: DS

Usage: DS holds the status of the last disk operation. It is a volatile variable. Each use triggers the reading of the disk status from the current disk device in usage. DS is coupled to the string variable DS\$ which is updated at the same time. Reading the disk status from a disk device automatically clears any error status on that device, so subsequent reads will return 0, if no other activity has since occurred.

Remarks: DS is a reserved system variable.

Example: Using DS

```
100 DOPEN#1,"DATA"  
110 IF DS<>0 THEN PRINT"COULD NOT OPEN FILE DATA":STOP
```

DSS

Format: **DSS**

Usage: **DSS** holds the status of the last disk operation in text form of the format: Code,Message,Track,Sector.

DSS is coupled to the numeric variable **DS**. It is updated when **DS** is used. DS\$ is set to **00,OK,00,00** if there was no error, otherwise it is set to a DOS error message (listed in the disk drive manuals).

Remarks: **DSS** is a reserved system variable.

Example: Using **DSS**

```
100 DOPENH1,"DATA"  
110 IF DS<>0 THEN PRINT DSS:STOP
```

DSAVE

Token: \$EF

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

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

filename is either a quoted string such as "DATA", or a string expression in brackets such as (F1\$). 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 not recommended to use this option on 1541 and 1571 drives, as they contain a "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 such as the 1541, 1571, or 1581.

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

Remarks: **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
```

DT\$

Format: **DT\$**

Usage: **DT\$** holds the current date and is updated before each usage from the RTC (Real-Time Clock). The string **DT\$** is formatted as: "DD-MON-YYYY", for example: "04-APR-2021".

Remarks: **DT\$** is a reserved system variable. For more information on how to set the Real-Time Clock, refer to the Configuration Utility section on page [4-12](#).

Example: Using **DT\$**

```
100 PRINT "TODAY IS: ";DT$
```

DVERIFY

Token: \$FE \$14

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

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

filename is either a quoted string such as "DATA", or a string expression in brackets such as **(F1\$)**.

drive drive # in dual drive disk units.

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

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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. **DVERIFY** exits either with the message **OK** or with **VERIFY ERROR**.

Example: Using **DVERIFY**

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

EDIT

Format: **EDIT <ON | OFF>**

Usage: **EDIT** switches the built-in editor either to text mode with **EDIT ON**, or to the BASIC program editor with**EDIT OFF**.

After power up or reset, the editor is initialised as BASIC program editor.

After setting the editor to text mode with **EDIT ON**, the differences to program mode are:

The editor does no tokenising/parsing. All text entered after a linenum-ber remains pure text, BASIC keywords such as **FOR** and **GOTO** are not converted to BASIC tokens, as they are whilst in program mode.

The line numbers are only used for text organisation, sorting, deleting, listing etc. When the text is saved to file with **DSAVE**, a sequential file (type **SEQ**) is written, not a program (**PRG**) file, which is how they're writ-ten whilst in program mode. Line numbers are not written to the file.

DLOAD in text mode can load only sequential files. Line numbers are automatically generated for editing purposes.

The mode of the editor can be recognised by looking at the prompt: In program mode, the prompt is **READY.**, whilst in text mode the prompt is **OK**.

Text mode affects entered lines with leading numbers only, lines with no line number are executed as BASIC commands, as usual.

Sequential files, created with the text editor, can be displayed (without loading them) on the screen by using **TYPE <filename>**.

Example: Using **EDIT**

```
ready.  
edit on  
  
ok.  
100 This is a simple text editor.  
dsave "example"  
  
ok.  
new  
  
ok.  
catalog  
  
0 "demoempty"    " 00 3d  
1 "example"      seq  
3159 blocks free  
  
ok.  
type "example"  
This is a simple text editor.  
  
ok.  
dload "example"  
  
loading  
  
ok.  
list  
  
1000 This is a simple text editor.  
  
ok.
```

EDMA

Token: \$FE \$21

Format: **EDMA** command, length, source, target [, sub, mod]

Usage: **EDMA** ("Extended Direct Memory Access") 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 (maximum = 65535).

source 28-bit address of read area or fill byte.

target 28-bit address of write area.

sub sub command (see chapter on DMA controller in the MEGA65 Book).

mod modifier (see chapter on DMA controller in the MEGA65 Book).

Remarks: **EDMA** can access the entire 256MB address range, using up to 28 bits for the addresses of the source and target.

Examples: Using **EDMA**

```
EDMA 0, $800, $F700, $8000000 :REM COPY SCALAR VARIABLES TO ATTIC RAM  
EDMA 3, 80*25, 32, 2048      :REM FILL SCREEN WITH BLANKS  
EDMA 0, 80*25, 2048, $8000800 :REM COPY SCREEN TO ATTIC RAM
```

EL

Format: **EL**

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

Remarks: **EL** is a reserved system variable.

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

Example: Using **EL**

```
10 TRAP 100
20 PRINT SQR(-1)      :REM PROVOKE ERROR
30 PRINT "AT LINE 30":REM HERE TO RESUME
40 END
100 IF ER>0 THEN PRINT ERR$(ER); " ERROR"
110 PRINT " IN LINE";EL
120 RESUME NEXT      :REM RESUME AFTER ERROR
```

ELLIPSE

Token: \$FE \$30

Format: **ELLIPSE** xc, yc, xr, yr [, flags , start, stop]

Usage: Draws an ellipse.

xc is the x coordinate of the centre in pixels

yc is the y coordinate of the centre in pixels

xr is the x radius of the ellipse in pixels

yr is the y radius of the ellipse in pixels

flags control filling, arcs and orientation of the zero radian (combs flag named after **retroCombs**). Default setting (zero) is: Don't fill, draw legs, start drawing at 3 'o clock.

Bit	Name	Value	Action if set
0	fill	1	Fill ellipse or arc with the current pen colour
1	legs	2	Suppress drawing of the legs of an arc
2	combs	4	Drawing (0 degree) starts at 12 'o clock

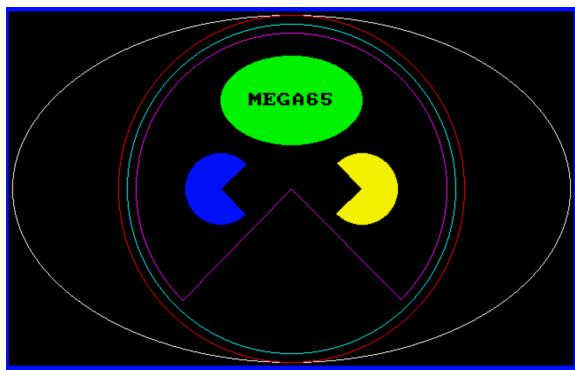
The units for the start- and stop-angle are degrees in the range of 0 to 360. The 0 radian starts at 3 o' clock and moves clockwise. The combs-flag shifts the 0 radian and the start position to the 12 o'clock position.

start start angle for drawing an elliptic arc.

stop stop angle for drawing an elliptic arc.

Remarks: **ELLIPSE** is used to draw ellipses on screens at various resolutions. If a full ellipse is to be drawn, start and stop should be either omitted or set both to zero (not 0 and 360). Drawing and filling of full ellipses is much faster, than using elliptic arcs.

Example: Using **ELLIPSE**



```
100 SX=2:DZ=3:WZ=320*S%:HZ=200*S% :REM SCREEN SETTINGS
110 CX% = WZ/2:CY% = HZ/2 :REM CENTRE AND RADII
120 RX% = WZ/2:RY% = HZ/2
130 SCREEN WZ,HZ,D% :REM OPEN SCREEN
140 ELLIPSE CX%,CY%,CX%-4,CY%-4
150 PEN2:CIRCLE CX%,CY%,RY%-4,2
160 PEN3:CIRCLE CX%,CY%,RY%-14,2
170 PEN4:CIRCLE CX%,CY%,RY%-24,0,135,45
180 PEN5:ELLIPSE CX%,CY%/2,RX%/4,RY%/4,1
190 PEN6:CIRCLE 120*S%,CY%,40,1,45,315
200 PEN7:CIRCLE 200*S%,CY%,40,1,225,135
210 PEN0:CHAR 34,CY%/2-8,2,2,2,"MEGAB5",\$3D000
220 GETKEY A% :REM WAIT FOR ANY KEY
230 SCREEN CLOSE :REM CLOSE GRAPHICS SCREEN
```

ELSE

Token: \$D5

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

Usage: **ELSE** is an optional part of an **IF** statement.

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

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

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

Remarks: There must be a colon before **ELSE**. There cannot be a colon or end-of-line after **ELSE**.

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

When the **true clause** does not use **BEGIN** and **BEND**, **ELSE** must be on the same line as **IF**.

Example: Using **ELSE**

```
100 REM ELSE
110 RED$=CHR$(28):BLACK$=CHR$(144):WHITE$=CHR$(5)
120 INPUT "ENTER A NUMBER";V
130 IF V<0 THENPRINT RED$;:ELSEPRINT BLACK$;
140 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
150 PRINT WHITE$
160 INPUT "END PROGRAM:(Y/N)";A$
170 IF A$="Y" THENEND
180 IF A$="N" THEN120:ELSE160
```

Using **ELSE** with **BEGIN** and **BEND**.

```
100 A = 0 : GOSUB 200
110 A = 1 : GOSUB 200
120 END
200 IF A = 0 THEN BEGIN
210 PRINT "HELLO"
220 BEND : ELSE BEGIN
230 PRINT "GOODBYE"
240 BEND
250 RETURN
```

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 nor close files. Also, variable definitions are still valid after **END**. The program may be continued with the **CONT** statement. After executing the last line of a program, **END** is automatically executed.

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.

There are 10 slots for storing instrument parameters, preset with the following default 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
20 VOL 9
30 TEMPO 30
40 PLAY "T904Q CDEFGAB U3T8 CDEFGAB L","T503Q H CGEQG T7 HGGEQG L"
```

ER

Format: ER

Usage: ER has the value of the most recent BASIC error that has occurred, or -1 if there was no error.

Remarks: ER is a reserved system variable.

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

Example: Using ER

```
10 TRAP 100
20 PRINT SQR(-1)      :REM PROVOKE ERROR
30 PRINT "AT LINE 30":REM HERE TO RESUME
40 END
100 IF ER>0 THEN PRINT ERR$(ER); " ERROR"
110 PRINT " IN LINE";EL
120 RESUME NEXT      :REM RESUME AFTER ERROR
```

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 such as "DATA", or a string expression in brackets such as (FI\$).

drive drive # in dual drive disk units.

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

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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 disk.

Remarks: ERASE **filename** is a synonym of SCRATCH **filename** and DELETE **filename**.

In direct mode the success and the number of erased files is printed. The second to last number from the message contains the number of successfully erased files,

Examples: Using ERASE

```
ERASE "DRM",U9 :REM ERASE FILE DRM ON UNIT 9
01, FILES SCRATCHED,01,00
ERASE "OLD*"   :REM ERASE ALL FILES BEGINNING WITH "OLD"
01, FILES SCRATCHED,04,00
ERASE "R*=PRG" :REM ERASE PROGRAM FILES STARTING WITH 'R'
01, FILES SCRATCHED,09,00
```

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
20 PRINT SQR(-1)      :REM PROVOKE ERROR
30 PRINT "AT LINE 30":REM HERE TO RESUME
40 END
100 IF ER>0 THEN PRINT ERR$(ER); " ERROR"
110 PRINT " IN LINE";EL
120 RESUME NEXT      :REM RESUME AFTER ERROR
```

EXIT

Token: \$FD

Format: EXIT

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

Remarks: In nested loops, **EXIT** exits only the current loop, and continues execution in an outer loop (if there is one).

Example: Using **EXIT**

```
1 REM EXIT
10 OPEN 2,8,0,"$"           : REM OPEN CATALOG
15 IF DS THEN PRINT DS$: STOP: REM CANT READ
20 GET#2,D$,D$              : REM DISCARD LOAD ADDRESS
25 DO                      : REM LINE LOOP
30  GET#2,D$,D$             : REM DISCARD LINE LINK
35  IF ST THEN EXIT         : REM END-OF-FILE
40  GET#2,L0,HI              : REM FILE SIZE BYTES
45  S=L0 + 256 * HI          : REM FILE SIZE
50  LINE INPUT#2, F$          : REM FILE NAME
55  PRINT S;F$              : REM PRINT FILE ENTRY
60 LOOP
65 CLOSE 2
```

EXP

Token: \$BD

Format: **EXP**(numeric expression)

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

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

Examples: Using **EXP**

```
PRINT EXP(1)  
2.71828183
```

```
PRINT EXP(0)  
1
```

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

FAST

Token: \$FE \$25

Format: **FAST** [speed]

Usage: Set CPU clock to 1MHz, 3.5MHz or 40MHz.

speed CPU clock speed where:

- **1** sets CPU to 1MHz.
- **3** sets CPU to 3MHz.
- Anything other than **1** or **3** sets the CPU to 40MHz.

Remarks: Although it's possible to call **FAST** with any real number, the precision part (the decimal point and any digits after it), will be ignored.

FAST is a synonym of **SPEED**.

FAST has no effect if **POKE 0,65** has previously been used to set the CPU to 40MHz.

Example: Using **FAST**

```
10 FAST      :REM SET SPEED TO MAXIMUM (40 MHZ)
20 FAST 1    :REM SET SPEED TO 1 MHZ
30 FAST 3    :REM SET SPEED TO 3.5 MHZ
40 FAST 3.5  :REM SET SPEED TO 3.5 MHZ
```

FGOSUB

Token: \$FE \$48

Format: **FGOSUB** numeric expression

Usage: Evaluates the given numeric expression, then calls (**GOSUBs**) the subroutine at the resulting line number.

Warning: Using this feature can break your program if **RENUMBER** is applied, as line numbers may change and the numeric expression will no longer address your intended line numbers.

Example: Using **FGOSUB**:

```
10 INPUT "WHICH SUBROUTINE TO EXECUTE 100,200,300";LI
20 FGOSUB LI      :REM HOPEFULLY THIS LINE # EXISTS
30 GOTO 10      :REM REPEAT
100 PRINT "AT LINE 100":RETURN
200 PRINT "AT LINE 200":RETURN
300 PRINT "AT LINE 300":RETURN
```

FGOTO

Token: \$FE \$47

Format: **FGOTO** numeric expression

Usage: Evaluates the given numeric expression, then jumps (**GOesTO**) to the resulting line number.

Warning: Using this feature can break your program if **RENUMBER** is applied, as line numbers may change and the numeric expression will no longer address your intended line numbers.

Example: Using **FGOTO**:

```
10 INPUT "WHICH LINE # TO EXECUTE 100,200,300";LI
20 FGOTO LI      :REM HOPEFULLY THIS LINE # EXISTS
30 END
100 PRINT "AT LINE 100":END
200 PRINT "AT LINE 200":END
300 PRINT "AT LINE 300":END
```

FILTER

Token: \$FE \$03

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

Usage: Sets the parameters for a SID sound filter.

sid 1: right SID, 2: left SID

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

Example: Using **FILTER**

```
10 PLAY "T7X103P9C"
15 SLEEP 0.02
20 PRINT "LOW PASS SWEEP":L=1:B=0:H=0:GOSUB 100
30 PRINT "BAND PASS SWEEP":L=0:B=1:H=0:GOSUB 100
40 PRINT "HIGH PASS SWEEP":L=0:B=0:H=1:GOSUB 100
50 GOTO 20
100 REM *** SWEEP ***
110 FOR F = 50 TO 1950 STEP 50
120 IF F >= 1000 THEN FF = 2000-F : ELSE FF = F
130 FILTER 1,FF,L,B,H,15
140 PLAY "X1"
150 SLEEP 0.02
160 NEXT F
170 RETURN
```

FIND

Token: \$FE \$2B

Format: **FIND /string/** [, line range]
FIND "string" [, line range]

Usage: **FIND** is an editor command that can only be used in direct mode. It searches a given line range (if specified), otherwise the entire BASIC program is searched. At each occurrence of the "find string" the line is listed with the string highlighted. **NO SCROLL** can be used to pause the output.

Remarks: Any un-shifted character that is not part of the string can be used instead of **/**.

However, using double quotes " as a delimiter has a special effect: The search text is not tokenised. **FIND "FOR"** will search for the three letters F, O, and R, not the BASIC keyword **FOR**. Therefore, it can find the word **FOR** in string constants or REM statements, but not in program code.

On the other hand, **FIND /FOR/** will find all occurrences of the BASIC keyword, but not the text "FOR" in strings.

Partial keywords cannot be searched. For example, **FIND /LOO/** will not find the keyword **LOOP**,

Example: Using **FIND**

```
READY.
LIST
10 REM PARROT COLOUR SCHEME
20 FONT 8 :REM SERIF
30 FOREGROUND 5 :REM GREEN
40 BACKGROUND 0 :REM BLACK
50 HIGHLIGHT 4,0 :REM SYSTEM PURPLE
60 HIGHLIGHT 14,1 :REM REM BLUE
70 HIGHLIGHT 7,2 :REM KEYWORD YELLOW

READY.
FIND /OLO/
10 REM PARROT COLOUR SCHEME

READY.
FIND /HIGHLIGHT/
50 HIGHLIGHT 4,0 :REM SYSTEM PURPLE
60 HIGHLIGHT 14,1 :REM REM BLUE
70 HIGHLIGHT 7,2 :REM KEYWORD YELLOW

READY.
■
```

FN

Token: \$A5

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

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

Example: Using **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
```

FONT

Token: \$FE \$46

Format: FONT <A | B | C>

Usage: FONT is used to switch between fonts, and the code pages PETSCII, and enhanced PETSCII. The enhanced PETSCII includes all ASCII symbols that are missing in the PETSCII code page, although the order is still PETSCII. The ASCII symbols are typed by pressing the keys in the table below, some of which also require the holding down of the  key. The codes for uppercase and lowercase are swapped compared to ASCII. The uppercase/graphics character set is not changed.

Code	Key	PETSCII	ASCII
\$5C	Pound	£	\ (backslash)
\$5E	Up Arrow (next to RESTORE)	↑	^ (caret)
\$5F	Left Arrow (next to 1)	↶	_ (underscore)
\$7B	MEGA + Colon	↶	{ (open brace)
\$7C	MEGA + Dot	↷	(pipe)
\$7D	MEGA + Semicolon	↷	} (close brace)
\$7E	MEGA + Comma	↶	~ (tilde)

Remarks: The additional ASCII characters provided by FONT A and B are only available while using the lowercase/uppercase character set.

Examples: Using FONT

```
FONT A :REM ASCII - ENABLE {}_~^
FONT B :REM LIKE A, WITH A SERIF FONT
FONT C :REM COMMODORE FONT (DEFAULT)
```

FOR

Token: \$81

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

Usage: **FOR** statements start a BASIC loop with an index variable.

index 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.

start is used to initialise the index.

end is checked at the end of an iteration, and determines whether another iteration will be performed, or if the loop will exit.

step defines the change applied to the index variable at the end of an iteration. 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 than or equal to **start**, whereas for negative increments **end** must be less than or equal to **start**.

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

Examples: 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=1 TO 20
40 M(I,J) = I + 100 * J
50 NEXT J,I
```

FOREGROUND

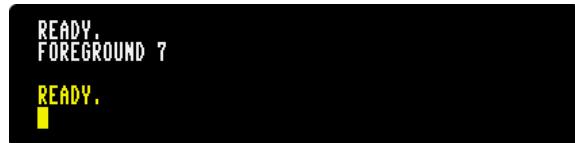
Token: \$FE \$39

Format: FOREGROUND colour

Usage: Sets the foreground colour (text colour) of the screen to the argument, which must be in the range of 0 to 31. Refer to the table under **BACKGROUND** on page [B-25](#) for the colour values and their corresponding colours.

Remarks: COLOR also has the ability to change the foreground colour.

Example: Using FOREGROUND



FORMAT

Token: \$FE \$37

Format: **FORMAT** diskname [,I id] [,D drive] [,U unit]

Usage: Used to format (or clear) a disk.

I The disk ID.

diskname is either a quoted string, e.g. "DATA" or a string expression in brackets, e.g. (DMS). The maximum length of **diskname** is 16 characters.

drive drive # in dual drive disk units.

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

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

Remarks: **FORMAT** and **HEADER** are aliases and call the same routine.

For new floppy disks which have not already been formatted in MEGA65 (1581) format, it is necessary to specify the disk ID with the I parameter. This switches the format command to low level format, which writes sector IDs and erases all contents. This takes some time, as every block on the floppy disk will be written.

If the I parameter is omitted, a quick format will be performed. This is only possible if the disk has already been formatted as a MEGA65 or 1581 floppy disk. A quick format writes the new disk name and clears the block allocation map, marking all blocks as free. The disk ID is not changed, and blocks are not overwritten, so contents may be recovered with **ERASE R**. You can read more about **ERASE** on page [B-101](#).

Examples: Using **FORMAT**

```
FORMAT "ADVENTURE",IDK : FORMAT DISK WITH NAME ADVENTURE AND ID DK  
FORMAT "ZORK-I",U9      : FORMAT DISK IN UNIT 9 WITH NAME ZORK-I  
FORMAT "DUNGEON",D1,U10: FORMAT DISK IN DRIVE 1 UNIT 10 WITH NAME DUNGEON
```

FRE

Token: \$B8

Format: **FRE(bank)**

Usage: Returns the number of free bytes for banks 0 or 1, or the ROM version if the argument is negative.

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. **FRE(1)** also triggers “garbage collection”, which is a process that collects strings in use at the top of the bank, thereby defragmenting string memory.

FRE(-1) returns the ROM version, a six-digit number of the form 92XXXX.

Example: Using **FRE**:

```
10 PM = FRE(0)
20 VM = FRE(1)
30 RV = FRE(-1)
40 PRINT PM;" FREE FOR PROGRAM"
50 PRINT VM;" FREE FOR VARIABLES"
60 PRINT RV;" ROM VERSION"
```

FREAD

Token: \$FE \$1C

Format: **FREAD#** channel, pointer, size

Usage: Reads **size** bytes from **channel** to memory starting at the 32-bit address **pointer**.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

Care must be taken not to overwrite memory that is used by the system or the interpreter.

It is recommended to use the **POINTER** statement for the pointer argument, and to compute the size parameter by multiplying the number of elements with the item size.

Type	Item Size
Byte Array	1
Integer Array	2
Real Array	5

Keep in mind that the **POINTER** function with a string argument does NOT return the string address, but the string descriptor. It is not recommended to use **FREAD** for strings or string arrays unless you are fully aware on how to handle the string storage internals.

Also, ensure that you always specify an index if you use an array. The start address of array **XY()** is **POINTER(XY(0))**. **POINTER(XY)** returns the address of the scalar variable **XY**.

Example: Using **FREAD**:

```
100 N=23
110 DIM B&(N),C&(N)
120 DOPEN#2,"TEXT"
130 FREAD#2,POINTER(B&(0)),N
140 DCLOSE#2
150 FOR I=0 TO N-1:PRINT CHR$(B&(I));:NEXT
160 FOR I=0 TO N-1:C&(I)=B&(N-I-I):NEXT
170 DOPEN#2,"REVERSE",N
180 FWRITE#2,POINTER(C&(0)),N
190 DCLOSE#2
```

FREEZER

Token: \$FE \$4A

Format: FREEZER

Usage: FREEZER calls the **FREEZER** program.

Remarks: Calling **FREEZER** via BASIC command is an alternative to the keypress of
RESTORE.

Examples: Using **FREEZER**

```
FREEZER :REM CALL FREEZER MENU
```

FWRITE

Token: \$FE \$1E

Format: FWRITE# channel, pointer, size

Usage: Writes **size** bytes to **channel** from memory starting at the 32-bit address **pointer**.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**.

It is recommended to use the **POINTER** statement for the pointer argument and compute the size parameter by multiplying the number of elements with the item size.

Refer to the **FREAD** item size table on page [B-116](#) for the item sizes.

Keep in mind that the **POINTER** function with a string argument does NOT return the string address, but the string descriptor. It is not recommended to use **FWRITE** for strings or string arrays unless you are fully aware on how to handle the string storage internals.

Also, ensure that you always specify an index if you use an array. The start address of array **XY()** is **POINTER(XY(0))**. **POINTER(XY)** returns the address of the scalar variable **XY**.

Example: Using **FWRITE**:

```
100 N=23
110 DIM B&(N),C&(N)
120 DOPEN#2,"TEXT"
130 FREAD#2,POINTER(B&(0)),N
140 DCLOSE#2
150 FOR I=0 TO N-1:PRINTCHR$(B&(I));:NEXT
160 FOR I=0 TO N-1:C&(I)=B&(N-1-I):NEXT
170 DOPEN#2,"REVERS",N
180 FWRITE#2,POINTER(C&(0)),N
190 DCLOSE#2
```

GCOPY

Token: \$FE \$32

Format: **GCOPY** x, y, width, height

Usage: **GCOPY** is used on graphic screens and copies the content of the specified rectangle with upper left position **x, y** and the **width** and **height** to the cut/copy/paste buffer.

The copied region can be inserted at any position with the command **PASTE**.

Remarks: The size of the rectangle is limited by the 1K size of the cut/copy/paste buffer. The memory requirement for a region is width * height * number of bitplanes / 8. It must not equal or exceed 1024 byte. For a 4-bitplane screen for example, a 45 x 45 region needs 1012.5 byte.

Example: Using **GCOPY** (see also **CUT**).

```
10 SCREEN 320,200,2
20 BOX 60,60,300,180,1 :REM DRAW A WHITE BOX
30 GCOPY 140,80,40,40 :REM COPY A 40 * 40 REGION
40 PASTE 10,10,40,40 :REM PASTE IT TO NEW POSITION
50 GETKEY A$           :REM WAIT FOR KEYPRESS
60 SCREEN CLOSE
```

GET

Token: \$A1

Format: **GET** variable

Usage: Gets the next character (or byte value of the next character) from the keyboard queue. If the variable being set to the character is of type string and the queue is empty, an empty string is assigned to it, otherwise a one character string is created and assigned instead. If the variable is of type numeric, the byte value of the key is assigned to it, otherwise zero will be assigned if the queue is empty. **GET** does not wait for keyboard input, so it's useful to check for key presses at regular intervals or in loops.

Remarks: **GETKEY** is similar, but waits until a key has been pressed.

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, variable [, variable ...]

Usage: Reads a single byte from the channel argument and assigns single character strings to string variables, or an 8-bit binary value to numeric variables. This is useful for reading characters (or bytes) from an input stream one byte at a time.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

Remarks: All values from 0 to 255 are valid, so **GET#** can also be used to read binary data.

Example: Using **GET#** to read a disk directory:

```
1 REM GET#
10 OPEN 2,8,0,"$"           : REM OPEN CATALOG
15 IF DS THEN PRINT DS$: STOP: REM CANT READ
20 GET#2,D$,D$              : REM DISCARD LOAD ADDRESS
25 DO                         : REM LINE LOOP
30 GET#2,D$,D$              : REM DISCARD LINE LINK
35 IF ST THEN EXIT           : REM END-OF-FILE
40 GET#2,LO,HI               : REM FILE SIZE BYTES
45 S=LO + 256 * HI           : REM FILE SIZE
50 LINE INPUT#2, F$           : REM FILE NAME
55 PRINT S;F$                : REM PRINT FILE ENTRY
60 LOOP
65 CLOSE 2
```

GETKEY

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

Format: **GETKEY** variable

Usage: Gets the next character (or byte value of the next character) from the keyboard queue. If the queue is empty, the program will wait until a key has been pressed. After a key has been pressed, the variable will be set and program execution will continue. When used with a string variable, a one character string is created and assigned. Otherwise if the variable is of type numeric, the byte value is assigned.

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 MEGA65 to C64-compatible -mode. If you're in direct mode, a security prompt **ARE YOU SURE?** is displayed, which must be responded with **Y** to continue. **\$Y\$58552** can be used to switch back to C65-mode.

Example: Using **GO64**:

```
GO64  
ARE YOU SURE?
```

GOSUB

Token: \$8D

Format: **GOSUB** line

Usage: **GOSUB** (GOto SUBroutine) 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 resumption of execution after the **GOSUB** statement, once a **RETURN** statement in the called subroutine is executed. Calls to subroutines via **GOSUB** may be nested, but the subroutines must always end with **RETURN**, otherwise a stack overflow may occur.

Remarks: Unlike other programming languages, BASIC 65 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 fewer digits to decode. The subroutines will also be found faster, since the search for subroutines often starts at the beginning of the program.

Example: Using **GOSUB**:

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

GOTO

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

Format: **GOTO** line
GO TO line

Usage: Continues program execution at the given BASIC line number.

Remarks: If the target **line** number is higher than the current line number, the search starts from the current line, proceeding to higher line numbers. If the target **line** number is lower, the search starts at the first **line** number of the program. It is possible to optimise the run-time speed of the program by grouping often used targets at the start (with lower line numbers).

GOTO (written as a single word) executes faster than **GO TO**.

Example: Using **GOTO**:

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

GRAPHIC

Token: \$DE

Format: **GRAPHIC CLR**

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

Once the graphics system has been cleared, commands such as **LINE**, **PALETTE**, **PEN**, **SCNCLR**, and **SCREEN** can be used to set graphic system parameters again.

Example: Using **GRAPHIC**:

```
100 REM GRAPHIC
110 GRAPHIC CLR      : REM INITIALISE
120 SCREEN DEF 1,1,1,2 : REM 640 X 400 X 2
130 SCREEN OPEN 1     : REM OPEN IT
140 SCREEN SET 1,1    : REM VIEW IT
150 PALETTE 1,0,0, 0,0 : REM BLACK
160 PALETTE 1,1,0,15,0 : REM GREEN
170 SCNCLR 0          : REM FILL SCREEN WITH BLACK
180 PEN 0,1            : REM SELECT PEN
190 LINE 50,50,590,350 : REM DRAW LINE
200 GETKEY A$         : REM WAIT FOR KEYPRESS
210 SCREEN CLOSE 1    : REM CLOSE SCREEN AND RESTORE PALETTE
```

HEADER

Token: \$F1

Format: HEADER diskname [,I id] [,D drive] [,U unit]

Usage: Used to format (or clear) a disk.

I The disk ID.

diskname is either a quoted string, e.g. "DATA" or a string expression in brackets, e.g. (DMS). The maximum length of **diskname** is 16 characters.

drive drive # in dual drive disk units.

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

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

Remarks: FORMAT and HEADER are aliases and call the same routine.

For new floppy disks which have not already been formatted in MEGA65 (1581) format, it is necessary to specify the disk ID with the I parameter. This switches the format command to low level format, which writes sector IDs and erases all contents. This takes some time, as every block on the floppy disk will be written.

If the I parameter is omitted, a quick format will be performed. This is only possible if the disk has already been formatted as a MEGA65 or 1581 floppy disk. A quick format writes the new disk name and clears the block allocation map, marking all blocks as free. The disk ID is not changed, and blocks are not overwritten, so contents may be recovered with ERASE R. You can read more about ERASE on page [B-101](#).

Examples: Using HEADER

```
HEADER "ADVENTURE",IDK : FORMAT DISK WITH NAME ADVENTURE AND ID DK  
HEADER "ZORK-I",U9      : FORMAT DISK IN UNIT 9 WITH NAME ZORK-I  
HEADER "DUNGEON",D1,U10: FORMAT DISK IN DRIVE 1 UNIT 10 WITH NAME DUNGEON
```

HELP

Token: \$EA

Format: HELP

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

Remarks: Displays BASIC errors. For errors related to disk I/O, the disk status variable **DS** or the disk status string **DSS\$** should be used instead.

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 hexadecimal representation of the argument. The argument must be in the range of 0-65535, corresponding to the hex numbers \$0000-\$FFFF.

Remarks: If real numbers are used as arguments, the fractional part will be ignored. In other words, real numbers will not be rounded.

Example: Using **HEX\$**:

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

HIGHLIGHT

Token: \$FE \$3D

Format: **HIGHLIGHT** colour [, mode]

Usage: Sets the colours used for highlighting. Different colours can be set for system messages, **REM** statements and BASIC 65 keywords.

colour is one of the first 16 colours in the current palette. Refer to page [B-25](#) for the colours in the default palette.

mode indicates what the colour will be used for.

- **0** system messages (the default mode)
- **1** **REM** statements
- **2** BASIC keywords

Remarks: The system messages colour is used when displaying error messages, and in the output of **CHANGE**, **FIND**, and **HELP**. The colours for **REM** statements and BASIC keywords are used by **LIST**.

Example: Using **HIGHLIGHT** to change the color of BASIC keywords to red.

```
LIST
10 REM *** THIS IS HELLO WORLD ***
20 PRINT "HELLO WORLD"
READY,
HIGHLIGHT 8,2
READY,
LIST
10 REM *** THIS IS HELLO WORLD ***
20 PRINT "HELLO WORLD"
READY.
```

IF

Token: \$8B

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

Usage: Starts a conditional execution statement.

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

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

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

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

Example: Using **IF**

```
1 REM IF
10 RED$=CHR$(28) : BLACK$=CHR$(144) : WHITE$=CHR$(5)
20 INPUT "ENTER A NUMBER";V
30 IF V<0 THEN PRINT RED$; : ELSE PRINT BLACK$;
40 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
50 PRINT WHITE$
60 INPUT "END PROGRAM: (Y/N)"; A$
70 IF A$="Y" THEN END
80 IF A$="N" THEN 20 : ELSE 60
```

IMPORT

Token: \$DD

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

Usage: The **IMPORT** command loads a BASIC program in text format and type **SEQ** into memory reserved for BASIC programs.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (F1\$).

drive drive # in dual drive disk units.

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

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

Remarks: The program is loaded into BASIC memory and converted from text to the tokenised form of **PRG** files. This enables loading of BASIC programs that were saved as plain text files as program listing. After loading, the program is re-linked and ready to be **RUN** or edited. It is possible to use **IMPORT** for merging a program text file from disk to a program already in memory. Each line read from the file is processed in the same way, as if typed from the user with the screen editor.

There is no **EXPORT** counterpart, because this function is already available. The sequence **DOPEN#1,"LISTING",W:CMD 1:LIST:DCLOSE#1** converts the program in memory to text and writes it to the file, that is named in the **DOPEN** statement.

Examples: Using **IMPORT**

```
IMPORT "APOCALYPSE"  
IMPORT "MEGA TOOLS",U9  
IMPORT (F1$),U(UNX)
```

INPUT

Token: \$85

Format: INPUT [prompt <, | ;>] variable [, variable ...]

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

prompt optional string expression to be printed as the prompt. 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 instead.

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

The input will be processed after the user presses .

Remarks: The user must take care to enter the correct type of input, so it matches the **variable list** types. Also, the number of input items must match the number of variables. A surplus of input items will be ignored, whereas too few input items trigger another request for input with the prompt ???. Typing non numeric characters for integer or real variables will produce a **TYPE MISMATCH ERROR**. Strings for string variables must be in double quotes ("") if they contain spaces or commas. Many programs that need a safe input routine use **LINE INPUT** and a custom parser, in order to avoid program errors by wrong user input.

Example: Using **INPUT**:

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

INPUT#

Token: \$84

Format: **INPUT#** channel, variable [, variable ...]

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

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

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.
LINE INPUT# may be used to read a whole record into a single string variable.

Sequential files, that can be read by **INPUT#** can be generated by programs with **PRINT#** or with the editor of the MEGA65. For example:

```
EDIT ON  
  
10 "CHUCK PEDDLE",1937,"ENGINEER OF THE 6502"  
20 "JACK TRAMIEL",1928,"FOUNDER OF CBM"  
30 "BILL MENSCH",1945,"HARDWARE"  
  
DSAVE "CBM-PEOPLE"  
EDIT OFF
```

Example: Using **INPUT#**:

```
10 DIM N$(100),B%(100),S$(100):
20 DOPEN#2,"CBM-PEOPLE":REM OPEN SEQ FILE
25 IF DS THEN PRINT DS$:STOP:REM OPEN ERROR
30 FOR I=0 TO 100
40 INPUT#2,N$(I),B%(I),S$(I)
50 IF ST AND 64 THEN 80:REM END OF FILE
60 IF DS THEN PRINT DS$:GOTO 80:REM DISK ERROR
70 NEXT I
80 DCLOSE#2
110 PRINT "READ";I+1;" RECORDS"
120 FOR J=0 TO I:PRINT N$(J):NEXT J
```

RUN
READ 3 RECORDS
CHUCK PEDDLE
JACK TRAMIEL
BILL MENSCH

TYPE "CBM-PEOPLE"
"CHUCK PEDDLE",1937,"ENGINEER OF THE 6502"
"JACK TRAMIEL",1928,"FOUNDER OF CBM"
"BILL MENSCH",1945,"HARDWARE"

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**.

An enhanced version of string search using pattern matching is used if the first character of the search string is a pound sign '#'. The pound sign is not part of the search but enables the use of the '.' (dot) as a wildcard character, which matches any character. The second special pattern character is the '*' (asterisk) character. The asterisk in the search string indicates that the preceding character may never appear, appear once, or repeatedly in order to be considered as a match.

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.

Examples: Using **INSTR**:

```
I = INSTR("ABCDEF","CD")      : REM I = 3
I = INSTR("ABCDEF","XY")      : REM I = 0
I = INSTR("RAIIIN","*AI*IN")  : REM I = 5
I = INSTR("ABCDEF","E.C.E")   : REM I = 3
I = INSTR(A$+B$,C$)
```

INT

Token: \$B5

Format: INT(numeric expression)

Usage: Returns the integer part of the argument. This function is **NOT** limited to the typical 16-bit integer range (-32768 to 32767), as it uses real arithmetic. The allowed range is therefore determined by the size of the real mantissa which is 32-bits wide (-2147483648 to 2147483647).

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

Examples: Using **INT**:

```
X = INT(1.9)      :REM X = 1
X = INT(-3.1)     :REM X = -3
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 controller 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**

KEY <ON | OFF>

KEY <LOAD | SAVE> filename

KEY number, string

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

KEY list current assignments.

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

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

KEY LOAD loads key definitions from file.

KEY SAVE saves key definitions to file.

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

Default assignments:

```
KEY  
KEY 1,CHR$(27)+"X"  
KEY 2,CHR$(27)+"0"  
KEY 3,"DIR"+CHR$(13)  
KEY 4,"DIR "+CHR$(34)+"*=PRG"+CHR$(34)+CHR$(13)  
KEY 5,"U"  
KEY 6,"KEY6"+CHR$(141)  
KEY 7,"L"  
KEY 8,"MONITOR"+CHR$(13)  
KEY 9,"Q"  
KEY 10,"KEY10"+CHR$(141)  
KEY 11,"W"  
KEY 12,"KEY12"+CHR$(141)  
KEY 13,CHR$(27)+"0"  
KEY 14,"U"+CHR$(27)+"0"  
KEY 15,"HELP"+CHR$(13)  
KEY 16,"RUN "+CHR$(34)+"*"+CHR$(34)+CHR$(13)
```

Remarks: The sum of the lengths of all assigned strings must not exceed 240 characters. Special characters such as RETURN or QUOTE are entered using their codes with the **CHR\$** function. Refer to **CHR\$** on page [B-47](#) for more information.

Examples: Using **KEY**:

KEY ON	:REM ENABLE FUNCTION KEYS
KEY OFF	:REM DISABLE FUNCTION KEYS
KEY	:REM LIST ASSIGNMENTS
KEY 2,"PRINT A"+CHR\$(14)	:REM ASSIGN PRINT PI TO F2
KEY SAVE "MY KEY SET"	:REM SAVE CURRENT DEFINITIONS TO FILE
KEY LOAD "ELEVEN-SET"	:REM LOAD DEFINITIONS FROM FILE

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 to or less than **n**, the resulting string will be identical to the argument string.

string a string expression.

n a numeric expression (0-255).

Remarks: Empty strings and zero length strings are legal values.

Example: Using LEFT\$:

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

LEN

Token: \$C3

Format: **LEN**(string)

Usage: Returns the length of a string.

string a string expression.

Remarks: There is no terminating character, as opposed to other programming languages such as C, which uses the NULL character. 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: Assigns values (or results of expressions) to variables.

Remarks: The **LET** statement is obsolete and not required. Assignment to variables can be done without using **LET**, but it has been left in BASIC 65 for backwards compatibility.

Examples: Using **LET**:

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

LINE

Token: \$E5

Format: LINE xbeg, ybeg [, xnext1, ynext1 ...]

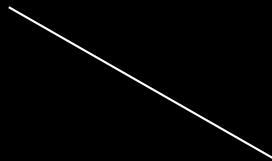
Usage: Draws a pixel at (xbeg/ybeg), if only one coordinate pair is given.

If more than one pair is defined, a line is drawn on the current graphics screen from the coordinate (xbeg/ybeg) to the next coordinate pair(s).

All currently defined modes and values of the graphics context are used.

Example: Using LINE:

```
1 REM SCREEN EXAMPLE 1
10 SCREEN 320,200,2      :REM SCREEN #0 320 X 200 X 2
20 PEN 1                  :REM DRAWING PEN COLOR 1 (WHITE)
30 LINE 25,25,295,175    :REM DRAW LINE
40 GETKEY A$              :REM WAIT FOR KEYPRESS
50 SCREEN CLOSE           :REM CLOSE SCREEN AND RESTORE PALETTE
```



LINE INPUT#

Token: \$E5 \$84

Format: **LINE INPUT#** channel, variable [, variable ...]

Usage: Reads one record per variable from an input device, (such as a disk drive) and assigns the read data to the variable. The records must be terminated by a **RETURN** character, which will not be copied to the string variable. Therefore, an empty line consisting of only the **RETURN** character will result in an empty string being assigned.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

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

Remarks: Only string variables or string array elements can be used in the variable list. Unlike other INPUT commands, **LINE INPUT#** does not interpret or remove quote characters in the input. They are accepted as data, as all other characters.

Records must not be longer than the input buffer, which is 160 characters.

Example: Using **LINE INPUT#**:

```
10 DIM N$(100)
20 DOPEN#2,"DATA"
30 FOR I=0 TO 100
40 LINE INPUT#2,N$(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"
```

LIST

Token: \$9B

Format: LIST [P] [line range]

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

line range consists of the first and/or last line to list, or a single line number. If the first number is omitted, the first BASIC line is assumed. If the second number is omitted, the last BASIC line is assumed.

Format: LIST [P] filename [,U unit]

Usage: Used to list a BASIC program directly from **unit**, which by default is 8.

Remarks: The optional parameter **P** enables page mode. After listing 24 lines, the listing will stop and display the prompt [MORE] at the bottom of the screen. Pressing **Q** quits page mode, while any other key triggers the listing of the next page.

LIST output can be redirected to other devices via **CMD**.

The keys **F9** and **F11**, or **Ctrl** **P** and **Ctrl** **V** scroll a BASIC listing on screen up or down.

Examples: 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
LIST "DEMO" :REM LIST FILE "DEMO"
LIST P :REM LIST PROGRAM IN PAGE MODE
LIST P "MURX" :REM LIST FILE "MURX" IN PAGE MODE
```

LOAD

Token: \$93

Format:

- LOAD** filename [, unit [, flag]]
- LOAD** "\$[pattern=type]" [, unit]
- LOAD** "\$\$[pattern=type]" [, unit]
- / filename [, unit [, flag]]

Usage: The first form loads a file of type **PRG** into memory reserved for BASIC programs.

The second form loads a directory into memory, which can then be viewed with **LIST** or **LSTP**. It is structured like a BASIC program, but file sizes are displayed instead of line numbers.

The third form is similar to the second one, but the files are numbered. This listing can be scrolled like a BASIC program with the keys **F9** or **F11**, edited, listed, saved or printed.

A filter can be applied by specifying a pattern or a pattern and a type. The asterisk matches the rest of the name, while the ? matches any single character. The type specifier can be a character of (P,S,U,R), that is Program, Sequential, User, or Relative file.

A common use of the shortcut symbol / is to quickly load **PRG** files. To do this:

1. Print a disk directory using either **DIR**, or **CATALOG**.
2. Move the cursor to the desired line.
3. type / in the first column of the line, and press **RETURN**.

After pressing **RETURN**, the listed file on the line with the leading / will be loaded. Characters before and after the file name double quotes ("") will be ignored. This applies to **PRG** files only.

filename is either a quoted string, e.g. "PROG", or a string expression.

The unit number is optional. If not present, the default disk device is assumed.

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

LOAD "/*" can be used to load the first **PRG** from the given **unit**.

LOAD "\$" can be be used to load the list of files from the given **unit**. When using **LOAD "\$"**, **LIST** can be used to print the listing to screen.

LOAD is implemented in BASIC 65 to keep it backwards compatible with BASIC V2.

The shortcut symbol / can only be used in direct mode.

By default the C64 uses **unit** 1, which is assigned to datasette tape recorders connected to the cassette port. However the MEGA65 uses **unit** 8 by default, which is assigned to the internal disk drive. This means you don't need to add ,8 to **LOAD** commands that use it.

Examples: Using **LOAD**

```
LOAD "APOCALYPSE"    :REM LOAD A FILE CALLED APOCALYPSE TO BASIC MEMORY  
LOAD "MEGA TOOLS",9 :REM LOAD A FILE CALLED "MEGA TOOLS" FROM UNIT 9 TO BASIC MEMORY  
LOAD "*",8,1         :LOAD THE FIRST FILE ON UNIT 8 TO RAM AS SPECIFIED IN THE FILE  
  
LOAD "$"             :REM LOAD WHOLE DIRECTORY - WITH FILE SIZES  
LOAD "$$"            :REM LOAD WHOLE DIRECTORY - SCROLLABLE  
LOAD "$$X*=P"        :REM DIRECTORY, WITH PRG FILES STARTING WITH 'X'
```

LOADIFF

Token: \$FE \$43

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

Usage: Loads an IFF file into graphics memory. The IFF (Interchange File Format) is supported by many different applications and operating systems. **LOADIFF** assumes that files contain bitplane graphics which fit into the MEGA65 graphics memory. Supported resolutions are:

Width	Height	Bitplanes	Colours	Memory
320	200	max. 8	max. 256	max. 64 K
640	200	max. 8	max. 256	max. 128 K
320	400	max. 8	max. 256	max. 128 K
640	400	max. 4	max. 16	max. 128 K

filename is either a quoted string such as "DATA", or a string expression in brackets such as (F1\$).

drive drive # in dual drive disk units.

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

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

Remarks: Tools are available to convert popular image formats to IFF. These tools are available on several operating systems, such as AMIGA OS, macOS, Linux, and Windows. For example, **ImageMagick** is a free graphics package that includes a tool called **convert**, which can be used to create IFF files in conjunction with the **ppmtoilbm** tool from the **Netpbm** package.

To use **convert** and **ppmtoilbm** for converting a JPG file to an IFF file on Linux:

```
convert <myImage.jpg> <myImage.ppm>
ppmtoilbm -aga <myImage.ppm> > <myImage.iff>
```

Example: Using **LOADIFF**

```
100 BANK128:SCNCLR
110 REM DISPLAY PICTURES IN 320 X 200 X 7 RESOLUTION
120 GRAPHIC CLR:SCREEN DEF 0,0,0,7:SCREEN OPEN 0:SCREEN SET 0,0
130 FORI=1TO7: READF$
140 LOADIFF(F$+".IFF"):SLEEP 4:NEXT
150 DATA ALIEN,BEAKER,JOKER,PICARD,PULP,TROOPER,RIPLEY
160 SCREEN CLOSE 0
170 PALETTE RESTORE
```

LOCK

Token: \$FE \$50

Format: **LOCK** filename/pattern [,D drive] [,U unit]

Usage: Used to lock files. The specified file or a set of files, that matches the pattern, is locked and cannot be deleted with the commands **DELETE**, **ERASE** or **SCRATCH**.

The command **UNLOCK** removes the lock.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

drive drive # in dual drive disk units.

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

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

Remarks: In direct mode the number of locked files is printed. The second to last number from the message contains the number of locked files,

Examples: Using **LOCK**

```
LOCK "DRM",U9 :REM LOCK FILE DRM ON UNIT 9  
03,FILES LOCKED,01,00  
LOCK "BS*"    :REM LOCK ALL FILES BEGINNING WITH "BS"  
03,FILES LOCKED,04,00
```

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 (**2.71828183**) as base, not 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
```

LOG10

Token: \$CE \$08

Format: **LOG10**(numeric expression)

Usage: Computes the value of the decimal logarithm of the argument. The decimal logarithm uses 10 as base.

Example: Using **LOG10**

```
PRINT LOG10(1)
0

PRINT LOG10(0)
?ILLEGAL QUANTITY ERROR

PRINT LOG10(5)
0.69897

PRINT LOG10(100);LOG10(10);LOG10(1);LOG10(0.1);LOG10(0.01)
2 1 0 -1 -2
```

LOOP

Token: \$EC

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

DO [<**UNTIL** | **WHILE**> logical expression]
. . . statements [**EXIT**]
LOOP [<**UNTIL** | **WHILE**> logical expression]

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

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

Examples: Using **DO** and **LOOP**

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

LPEN

Token: \$CE \$04

Format: LPEN(coordinate)

Usage: This function requires the use of a CRT monitor (or TV), and a light pen. It will not work with an 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, therefore **LPEN(0)** only returns even numbers. A bright background colour is needed to trigger the light pen. The **COLLISION** statement may be used to enable an interrupt handler.

Example: Using **LPEN**

```
PRINT LPEN(0),LPEN(1) :REM PRINT LIGHT PEN COORDINATES
```

MEM

Token: \$FE \$23

Format: **MEM mask4,mask5**

Usage: **mask4** and **mask5** are byte values, that are interpreted as mask of 8 bits. Each bit set to 1 reserves an 8K segment of memory in bank 4 for the first argument and in bank 5 for the second argument..

bit	memory segment
0	\$0000 - \$1FFF
1	\$2000 - \$3FFF
2	\$4000 - \$5FFF
3	\$6000 - \$7FFF
4	\$8000 - \$9FFF
5	\$A000 - \$BFFF
6	\$C000 - \$DFFF
7	\$E000 - \$FFFF

Remarks: After reserving memory with **MEM** the graphics library will not use the reserved areas, so it can be used for other purposes. Access to bank 4 and 5 is possible with the commands **PEEK**, **WPEEK**, **POKE**, **WPOKE** and **EDMA**.

If a graphics screen cannot be opened, because the remaining memory is not sufficient, the program stops with a **?OUT OF MEMORY ERROR**.

Example: Using **MEM**

```
10 MEM 1,3      :REM RESERVE $40000 - $41FFF AND $50000 - $53FFF
20 SCREEN 320,200 :REM SCREEN WILL NOT USE RESERVED SEGMENTS
40 EDMA 3,$2000,0,$4000:REM FILL SEGMENT WITH ZEROES
```

MERGE

Token: \$E6

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

Usage: **MERGE** loads a BASIC program file from disk and appends it to the program in memory.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (F1\$).

drive drive # in dual drive disk units.

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

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

Remarks: The load address, which is stored in the first two bytes of the file is ignored. The loaded program does not replace a program in memory (which is what **DLOAD** does), but is appended to a program in memory. After loading, the program is re-linked and ready to run or edit.

It is the user's responsibility to ensure that there are no line number conflicts among the program in memory and the merged program. The first line number of the merged program must be greater than the last line number of the program in memory.

Example: Using **MERGE**

```
DLOAD "MAIN PROGRAM"  
MERGE "LIBRARY"
```

MID\$

Token: \$CA

Format: **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 (1-255).

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-6
MEGA+65
```

MKDIR

Token: \$FE \$51

Format: **MKDIR** dirname ,**L** size [,**U** unit]

Usage: Make (create) a subdirectory on a floppy or D81 disk image.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

MKDIR can only be used on units, managed by CBDOS. These are the internal floppy disk drive and SD-Card images of **D81** type. The command cannot be used on external drives connected to the serial IEC bus.

The **size** parameter specifies the number of tracks, to be reserved for the subdirectory, with one track = 40 sectors at 256 byte. The first track of the reserved range is used as directory track for the subdirectory.

The minimum size is 3 tracks, the maximum 38 tracks. There must be a contiguous region of empty tracks on the floppy (D81 image), that is large enough for the creation of the subdirectory. The error message DISK FULL is reported, if there isn't such a region.

Several subdirectories may be created as long as there are enough empty tracks.

After successful creation of the subdirectory an automatic **CHDIR** into this subdirectory is performed.

CHDIR "/" changes back to the root directory.

Examples: Using **MKDIR**

```
MKDIR "SUBDIR",L5 :REM MAKE SUBDIRECTORY WITH 5 TRACKS
DIR
0 "SUBDIR" 1D
160 BLOCKS FREE.
```

MOD

Token: \$NN

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

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

Remarks: In other programming languages such as C, this function is implemented as an operator (%). In BASIC 65 it is implemented as a 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, the assembly language they use, and their architectures. More information on the **MONITOR** is available in the *MEGA65 Book*, ?? (??).

To exit the monitor press **X**.

Help text can be displayed with either **?** or **H**.

Example: Using **MONITOR**

MONITOR

```
BS MONITOR COMMANDS:ABCDEFHJMRTUXe,>?$$*&Z!ESV
PC SR AC XR YR ZR BP SP NUEBDIZC
; 00FFA2 00 00 00 00 00 00 01F8 -----
ASSEMBLE - A ADDRESS MNEMONIC OPERAND
BITMAPS - B [FROM]
COMPARE - C FROM TO WIIN
DISASSEMBLE - D [FROM [TO]]
FILL - F FROM TO FILLBYTE
GO - G [ADDRESS]
HUNT - H FROM TO (STRING OR BYTES)
JSR - J ADDRESS
LOAD - L FILENAME [UNIT [ADDRESS]]
MEMORY - M [FROM [TO]]
REGISTERS - R
SAVE - S FILENAME UNIT FROM TO
TRANSFER - T FROM TO TARGET
VERIFY - U FILENAME [UNIT [ADDRESS]]
EXIT - X
(DOT) - . ADDRESS MNEMONIC OPERAND
(X GREATER) - > ADDRESS BYTE SEQUENCE
; SEMICOLON) - ; REGISTER CONTENTS
@DOS - @ IDOS COMMAND
?HELP - ?
```

MOUNT

Token: \$FE \$49

Format: **MOUNT** filename [,U unit]

Usage: Mount a floppy image file of type **D81** from SD-Card to unit 8 (default) or unit 9.

If no argument is given, **MOUNT** assigns the real floppy drive of the MEGA65 to unit 8.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

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

Remarks: **MOUNT** can be used either in direct mode or in a program. It searches the file on the SD-card and mounts it, as requested, on unit 8 or 9. After mounting the floppy image can be used as usual with all DOS commands.

Examples: Using **MOUNT**

```
MOUNT "APOCALYPSE.D81" ;REM MOUNT IMAGE TO UNIT 8
MOUNT "BASIC.D81",U9    ;REM MOUNT IMAGE TO UNIT 9
MOUNT (FI$),U(UN%)     ;REM MOUNT WITH VARIABLE ARGUMENTS
MOUNT                  ;REM SELECT REAL FLOPPY DRIVE
```

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).

MOUSE OFF 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.

Examples: 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** number, position

Usage: Moves a sprite on screen. Each **position** argument consists of two 16-bit values, which specify either an absolute coordinate, a relative coordinate, an angle, or a speed. The value type is determined by a prefix:

- **+value** relative coordinate: positive offset.
- **-value** relative coordinate: negative offset.
- **#value** speed.

If no prefix is given, the absolute coordinate or angle is used.

Therefore, the position argument can be used to either:

- set the sprite to an absolute position on screen.
- specify a displacement relative from the current position.
- trigger a relative movement from a specified position.
- describe movement with an angle and speed starting from the current position.

MOVSPR number, position is used to set the sprite immediately to the position or, in the case of an angle#speed argument, describe its further movement.

Format: **MOVSPR** number, start-position **TO** end-position, speed

Usage: Places the sprite at the start position, defines the destination position, and the speed of movement. The sprite is placed at the start position, and will move in a straight line to the destination at the given speed. Coordinates must be absolute or relative. The movement is controlled by the BASIC interrupt handler and happens concurrently with the program execution.

number sprite number (0-7).

position x,y | xrel,y | x,yrel | xrel,yrel | angle#speed.

x absolute screen coordinate pixel.

y absolute screen coordinate pixel.

xrel relative screen coordinate pixel.

yrel relative screen coordinate pixel.

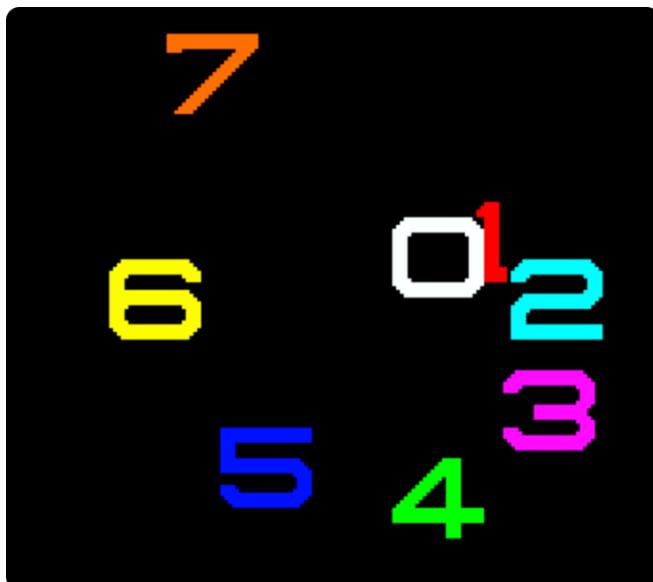
angle compass direction for sprite movement [degrees]. 0: up, 90: right, 180: down, 270: left, 45 upper right, etc.

speed speed of movement, configured as a floating point number in the range of 0.0-127.0, in pixels per frame. PAL has 50 frames per second whereas NTSC has 60 frames per second. A speed value of 1.0 will move the sprite 50 pixels per second in PAL mode.

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

Example: Using **MOVSPR**:

```
100 CLR:SCNCLR:SPRITECLR  
110 BLOAD "DEMO_SPRITES1",B0,P1536  
130 FOR I=0 TO 7: C=I+1:SP=0.07*(I+1)  
140 MOVSPRI, 160,120  
145 MOVSPRI,45*IHSP  
150 SPRITEI,I,C,,0,0  
160 NEXT  
170 SLEEP 3  
180 FOR I=0 TO 7:MOVSPR I,0#0:NEXT
```



NEW

Token: \$A2

Format: **NEW**

NEW RESTORE

Usage: Resets all BASIC parameters to their default values. Since **NEW** resets parameters and pointers, (but does not overwrite the address range of a BASIC program that was in memory), it is possible to recover the program. If there were no **LOAD** operations, or editing performed after **NEW**, the program can be restored with the **NEW RESTORE**.

Examples: 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: Marks the end of the BASIC loop associated with the given index variable. When a BASIC loop is declared with **FOR**, it must end with **NEXT**.

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.

start value to initialise the index with.

end is checked at the end of an iteration, and determines whether another iteration will be performed, or if the loop will exit.

step defines the change applied to the index variable at the end of every iteration. 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 omitted, 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, whereas real operands are converted to a signed 16-bit integer (losing precision). Logical operands are converted to a 16-bit integer, using \$FFFF (decimal -1) for TRUE, and \$0000 (decimal 0) for FALSE.

Expression	Result
NOT 0	1
NOT 1	0

Remarks: The result is of type integer.

Examples: Using NOT

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

In most cases, NOT is 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, such as **COLOR**, **KEY**, and **MOUSE**.

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

Examples: 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 number [, line number ...]
ON expression **GOTO** line number [, line number ...]
keyword **ON**

Usage: **ON** calls either a computed **GOSUB** or **GOTO** statement. Depending on the result of the expression, the target for **GOSUB** and **GOTO** is chosen from the table of line addresses at the end of the statement.

When used as a secondary keyword, **ON** is used in combination with primary keywords, such as **COLOR**, **KEY**, and **MOUSE**.

expression is a positive numeric value. Real values are converted to integer (losing precision). Logical operands are converted to a 16-bit integer, using \$FFFF (decimal -1) for TRUE, and \$0000 (decimal 0) for FALSE.

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

An expression result of zero, or a result that is greater than the number of target lines will not do anything, and the program will continue 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** channel, first address [, secondary address [, filename]]

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

channel number, where:

- **1 <= channel <= 127** line terminator is CR.
- **128 <= channel <= 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 reserved 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 **DOPEN#**, since the **filename** for **OPEN** includes all file attributes, for example: "0:DATA,\$,W".

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

If the first character of the filename is an at sign '@', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

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,$,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 (losing precision). Logical operands are converted to a 16-bit integer using \$FFFF (decimal -1) for TRUE, and \$0000 (decimal 0), for FALSE.

Expression	Result
0 OR 0	0
0 OR 1	1
1 OR 0	1
1 OR 1	1

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

Example: Using **OR**

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

In most cases, **OR** is used in **IF** statements.

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

PAINT

Token: \$DF

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

Usage: Performs a flood fill of an enclosed graphics area using the current pen colour.

x, y is a coordinate pair, which must lie inside the area to be painted.

mode specifies the paint mode:

- **0** The colour of pixel (x,y) defines the colour, which is replaced by the pen colour.
- **1** The **region border colour** defines the region to be painted with the pen colour.
- **2** Paint the region connected to pixel (x,y).

region border colour defines the colour index for mode 1.

Example: Using PAINT

```
10 SCREEN 320,200,2      :REM OPEN SCREEN
20 PALETTE 0,1,10,15,10 :REM COLOUR 1 TO LIGHT GREEN
30 PEN 1                 :REM SET DRAWING PEN (PEN 0) TO LIGHT GREEN (1)
40 LINE 160,0,240,100    :REM 1ST. LINE
50 LINE 240,100,80,100   :REM 2ND. LINE
60 LINE 80,100,160,0     :REM 3RD. LINE
70 PAINT 160,10          :REM FILL TRIANGLE WITH PEN COLOUR
80 GETKEY A&              :REM WAIT FOR KEY
90 SCREEN CLOSE           :REM END GRAPHICS
```

PALETTE

Token: \$FE \$34

Format: **PALETTE** screen, colour, red, green, blue
PALETTE COLOR colour, red, green, blue
PALETTE RESTORE

Usage: **PALETTE** 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 (0-3).

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 REM CHANGE SYSTEM COLOUR INDEX
20 REM --- INDEX 9 (BROWN) TO (DARK BLUE)
30 PALETTE COLOR 9,0,0,7
```

```
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 2             :REM SET DRAWING PEN (PEN 0) TO BLUE (2)
100 LINE 160,0,240,100 :REM 1ST. LINE
110 LINE 240,100,80,100 :REM 2ND. LINE
120 LINE 80,100,160,0  :REM 3RD. LINE
130 PAINT 160,10,0,2   :REM FILL TRIANGLE WITH BLUE (2)
140 GETKEY K$         :REM WAIT FOR KEY
150 SCREEN CLOSE 1    :REM END GRAPHICS
```

PASTE

Token: \$E3

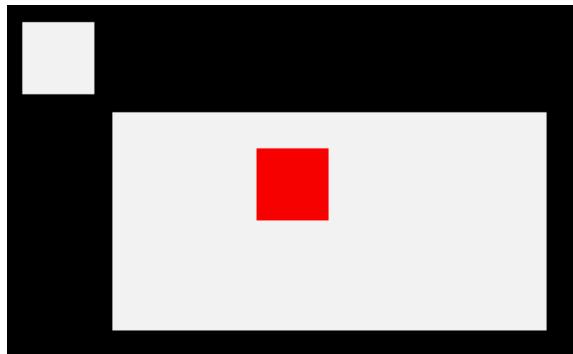
Format: PASTE x, y, width, height

Usage: PASTE is used on graphic screens and pastes the content of the cut/copy/paste buffer into the screen. The arguments upper left position **x**, **y** and the **width** and **height** specify the paste position on the screen.

Remarks: The size of the rectangle is limited by the 1K size of the cut/copy/paste buffer. The memory requirement for region is width * height * number of bitplanes / 8. It must not equal or exceed 1024 byte. For a 4-bitplane screen for example, a 45 x 45 region needs 1012.5 byte.

Example: Using PASTE

```
10 SCREEN 320,200,2
20 BOX 60,60,300,180,1 :REM DRAW A WHITE BOX
30 PEN 2 :REM SELECT RED PEN
40 CUT 140,80,40,40 :REM CUT OUT A 40 * 40 REGION
50 PASTE 10,10,40,40 :REM PASTE IT TO NEW POSITION
60 GETKEY A$ :REM WAIT FOR KEYPRESS
70 SCREEN CLOSE
```



PEEK

Token: \$C2

Format: PEEK(address)

Usage: Returns an unsigned 8-bit value (byte) from **address**.

If the address is in the range of \$0000 to \$FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to \$10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

Remarks: Banks 0-127 give access to RAM or ROM banks. Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using PEEK

```
10 BANK 128      :REM SELECT SYSTEM BANK
20 L = PEEK($02F8) :REM USR JUMP TARGET LOW
30 H = PEEK($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 of the graphic pen.

pen pen number (0-2):

- **0** drawing pen (default, if only single parameter provided).
- **1** off bits in jam2 mode.
- **2** currently unused.

colour palette index. Refer to the table under **BACKGROUND** on page [B-25](#) for the colour values and their corresponding colours.

Remarks: The colour selected by **PEN** will be used by all graphic/drawing commands that follow it. If you intend to set the drawing **pen** 0 to a colour, you can omit the first parameter, and only provide the **colour** parameter.

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, 0 :REM 0 = BLACK
60 PALETTE 1,1, 15, 0, 0 :REM 1 = RED
70 PALETTE 1,2, 0, 0,15 :REM 2 = BLUE
80 PALETTE 1,3, 0,15, 0 :REM 3 = GREEN
90 PEN 1             :REM SET DRAWING PEN (PEN 0) TO RED (1)
100 LINE 160,0,240,100 :REM DRAW RED LINE
110 PEN 2             :REM SET DRAWING PEN (PEN 0) TO BLUE (2)
120 LINE 240,100,80,100 :REM DRAW BLUE LINE
130 PEN 3             :REM SET DRAWING PEN (PEN 0) TO GREEN (3)
140 LINE 80,100,160,0   :REM DRAW GREEN LINE
150 GETKEY K$         :REM WAIT FOR KEY
160 SCREEN CLOSE 1    :REM END GRAPHICS
```

PIXEL

Token: \$CE \$0C

Format: **PIXEL**(*x*, *y*)

Usage: Returns the colour of a pixel at the given position.

x absolute screen coordinate.

y absolute screen coordinate.

PLAY

Token: \$FE \$04

Format: **PLAY** {[string1, string2, string3, string4, string5, string6]}

Usage: **PLAY** without any arguments will cause all voices to be silenced, and all of BASIC's music-system variables to be reset (E.g. **TEMPO**).

PLAY can be followed by up to six comma-separated string arguments, where each argument provides the sequence of notes and directives to be played on a specific voice on the two available SID chips, allowing for up to 6-channel polyphony.

Note that **PLAY** makes use of SID1 (for voices 1 to 3) and SID3 (for voices 4 to 6) of the 4 SID chips of the system. Also note that, by default, SID1 and SID2 are slightly right-biased and SID3 and SID4 are slightly left-biased (in terms of stereo sound).

A musical note is a character (A, B, C, D, E, F, or G), which may be preceded by an optional modifier.

Possible modifiers are:

Character	Effect
#	Sharp
\$	Flat
.	Dotted
H	Half Note
I	Eighth Note
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	Instrument Envelope	0 - 9
U	Volume	0 - 9
X	Filter	0 - 1
M	Modulation	0 - 9
P	Portamento	0 - 9
L	Loop	N/A

The modulation directive will modulate your note by the magnitude you specify (1-9), or use 0 to not use it.

Similarly, the portamento directive will gently slide between consecutive notes at the speed you specify (1-9), or use 0 to not use it. Note that the gate-off behaviour of notes is disabled while portamento is enabled, and to re-enable it, you must disable portamento (P0).

Add an **L** directive (no argument needed) at the end of your string if you would like it to loop back to the beginning of your string upon completion.

You have a lot of flexibility on which voice channels you choose to play your melodies on. For instance, you may decide to use only voice 1 and voice 4 for your melody, and spare the other channels for melody-based sound effects (for simple one-shot sound effects, consider the **SOUND** command instead). Just skip the voices you're not using with **PLAY**, by leaving those arguments empty:

```
PLAY "04EDCDEEERL","","02CGEGCGEGL"
```

You can even call **PLAY** again to use the aforementioned unused channels, to play another melody alongside your first melody. For example, using voice 2 and voice 5 this time:

```
PLAY ,,"05T2IGAGFEDCEG06.QCL","","03T2.QG.B 04IC036E.QCL"
```

If you wish to assess whether a melody is playing on a voice channel, you can find out by checking the value returned from **RPLAY(voice)**, where the voice parameter is a value from 1 to 6, indicating the voice channel. It will return either 1 (playing), or 0 (not playing).

One caveat to be aware of is that BASIC strings have a maximum length of 255 bytes. If your melody needs to exceed this length, consider breaking up your melody into several strings, then use **RPLAY(voice)** to assess when your first string has finished and then play the next string.

Instrument envelope slots may be modified by using the **ENVELOPE** statement. The default settings for the envelopes are on page [B-99](#).

Remarks: The **PLAY** statement makes use of an interrupt driven routine that starts parsing the string and playing the melody. Program execution continues with the next statement, and will not block until the melody has finished.

The 6 voice channels used by the **PLAY** command (on SID1+SID3) are distinct to the 6 channels used by the **SOUND** command (on SID2+SID4).

Example: Using **PLAY**

```
5 REM *** SIMPLE LOOPING EXAMPLE ***
10 ENVELOPE 9,10,5,10,5,0,300
20 VOL 8
30 TEMPO 30
40 PLAY "05T9HCIDCDEHCG IGAGFEFDEWCL", "02T0QCGEGC6EG DBGB CGEGL"
```

```
5 REM *** MODULATION + PORTAMENTO EXAMPLE ***
10 TEMPO 20
20 M$ = "M5 T205P00D P5FP0RP5QG .A1HQA HQQE.C IDQE HFQD .D1MCQD HEQHCQ04HA"
30 M$ = M$ + "05QDHFQG.A1HQA HQQE.C IDQEFEDE#C04B05HC D04AFD POR L"
40 B$ = "T0QR02H.D.F.C01.A.#A.G.A QAI02AGFE H.D.F.C01.A.#A.A02 .D DL"
50 PLAY M$,B$
```

POINTER

Token: \$CE \$0A

Format: **POINTER(variable)**

Usage: Returns the current address of a variable or an array element as a 32-bit pointer. For string variables, it is the address of the string descriptor, not the string itself. The string descriptor consists of three bytes (length, string address low, string address high).

Remarks: The address values of arrays and their elements are constant while the program is executing.
However, the addresses of strings (not their descriptors) may change at any time due to "garbage collection".

Example: Using **POINTER**

```
10 BANK 0          :REM SCALARS ARE IN BANK 0
20 H$="HELLO"     :REM ASSIGN STRING TO H$
30 P=POINTER(H$)  :REM GET DESCRIPTOR ADDRESS
40 PRINT "DESCRIPTOR AT: $";HEX$(P)
50 L=PEEK(P):SP=WPEEK(P+1) :REM LENGTH & STRING POINTER
60 PRINT "LENGTH = ";L    :REM PRINT LENGTH
70 BANK 1          :REM STRINGS ARE IN BANK 1
80 FOR I%=0 TO L-1:PRINT PEEK(SP+I%);:NEXT:PRINT
90 FOR I%=0 TO L-1:PRINT CHR$(PEEK(SP+I%));:NEXT:PRINT

RUN
DESCRIPTOR AT: $FD75
LENGTH = 5
72 69 76 76 79
HELLO
```

POKE

Token: \$97

Format: **POKE** address, value [, value ...]

Usage: Writes one or more bytes into memory or memory mapped I/O, starting at **address**.

If the address is in the range of \$0000 to \$FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to \$10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

If **value** is in the range of 0-255, this is poked into memory, otherwise the low byte of value is used. So a command like **POKE AD,V AND 255** can be written as **POKE AD,V** because **POKE** uses the lowy byte anyway.

Remarks: The address is incremented for each data byte, so a memory range can be written to with a single **POKE**.

Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **POKE**

```
10 BANK 128      :REM SELECT SYSTEM BANK
20 POKE $02F8,0,24 :REM SET USR VECTOR TO $1800
```

POLYGON

Token: \$FE \$2F

Format: **POLYGON** x, y, xrad, yrad, sides [{, drawssides, subtend, angle, solid}]

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.

sides number of polygon sides.

drawssides sides to draw.

subtend draw line from centre to start (1).

angle start angle.

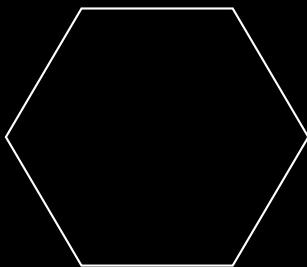
solid fill (1) or outline (0).

Remarks: A regular polygon is both isogonal and isotonal, meaning all sides and angles are alike.

Example: Using **POLYGON**

```
100 SCREEN 320,200,1      :REM OPEN 320 x 200 SCREEN
110 POLYGON 160,100,40,40,6 :REM DRAW HONEYCOMB
120 GETKEY @$              :REM WAIT FOR KEY
130 SCREEN CLOSE            :REM CLOSE GRAPHICS SCREEN
```

Results in:



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 anticlockwise limit.

A value greater than 255 indicates that the fire button is also being pressed.

Remarks: Analogue paddles are noisy and inexact. The range may be less than 0-255 and there could be some jitter in the values returned from **POT**.

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   : REM 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, depending on the argument type. For user controlled formatting, see **PRINT USING**.

The following argument types are evaluated:

- **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 in either fixed point form (typically 9 digits), or scientific form if the value is outside the range of 0.01 to 999999999.
- **string** the string may consist of printable characters and control codes. Printable characters are printed at the cursor position, while control codes are executed.
- , a comma acts as a tabulator.
- ; a semicolon acts as a separator between arguments of the list. Other than the comma character, it does not insert any additional characters. A semicolon at the end of the argument list suppresses the automatic return (carriage return) character.

Remarks: The **SPC** and **TAB** functions may be used in the argument list for positioning. **CMD** 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 formatted values to the device assigned to **channel**. Standard formatting is used, depending on the argument type. For user controlled formatting, see **PRINT# USING**.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**.

The following argument types are evaluated:

- **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 in either fixed point form (typically 9 digits), or scientific form if the value is outside the range of 0.01 to 999999999.
- **string** may consist of printable characters and control codes. Printable characters are printed at the cursor position, while control codes are executed.
- , a comma acts as a tabulator.
- ; a semicolon acts as a separator between arguments of the list. Other than the comma character, it does not insert any additional characters. A semicolon at the end of the argument list suppresses the automatic return (carriage return) character.

Remarks: The **SPC** and **TAB** functions are not suitable for devices other than the screen.

Example: Using **PRINT#** to write a file to drive 8:

```
10 DOPEN#2,"TABLE",W,U8
20 FOR I=1 TO 10 : REM START LOOP
30 PRINT#2,I,I*I,SQR(I)
40 NEXT
50 DCLOSE#2
```

You can confirm that the file '**TABLE**' has been written by typing **DIR "TABLE"**, and then view the contents of the file by typing **TYPE "TABLE"**.

PRINT USING

Token: \$98 \$FB or \$99 \$FB

Format: PRINT[# channel,] USING format; argument

Usage: Parses the **format** string and evaluates the argument. The argument can be either a string or a numeric value. The format of the resulting output is directed by the **format** string.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**. If no channel is specified, the output goes to the screen.

format string variable or a string constant which defines the rules for formatting. When using a number as the **argument**, formatting can be done in either CBM style, providing a pattern such as ####.## or in C style using a <width.precision> specifier, such as %3D %7.2F %4X .

argument the number to be formatted. If the argument does not fit into the format e.g. trying to print a 4 digit variable into a series of three hashes (###), asterisks will be used instead.

Remarks: The format string is applied for one argument only, but it is possible to append more with **USING format;argument** sequences.

argument 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. If the first character of the format string is an equals '=' sign, the argument string is centered. If the first character of the format string is a greater than '>' sign, the argument string is right justified.

Examples: Using **PRINT# USING**

```
PRINT USING "####.###";#, USING "[%6.4F] ";SQR(2)
3.14 [1.4142]

PRINT USING " < # # # > ";12*31
< 3 7 2 >

PRINT USING "#####"; "ABCDE"
ABC

PRINT USING ">#####"; "ABCDE"
CDE

PRINT USING "ADDRESS:$%4X";65000
ADDRESS:$FDE8

A$="#####,#####.##":PRINT USING A$;1E8/3
33,333,333.3
```

RCOLOR

Token: \$CD

Format: **RCOLOR**(colour source)

Usage: Returns the current colour index for the selected colour source.

Colour sources are:

- **0** background colour (VIC \$D021).
- **1** text colour (\$F1).
- **2** highlight colour (\$2D8).
- **3** border colour (VIC \$D020).

Example: Using **RCOLOR**

```
10 C = RCOLOR(3) : REM C = colour index of border colour
```

RCURSOR

Token: \$FE \$42

Format: **RCURSOR** {colvar, rowvar}

Usage: Returns the current cursor column and row.

Remarks: The row and column values start at zero, where the left-most column is zero, and the top row is zero.

Example: Using **RCURSOR**

```
100 CURSOR ON,20,10  
110 PRINT "[HERE]";  
120 RCURSOR X,Y  
130 PRINT " COL:",X;" ROW:",Y
```

RUN

```
[HERE] COL: 26  ROW: 10
```

READ

Token: \$87

Format: **READ** variable [, variable ...]

Usage: Reads values from program source into variables.

variable list Any legal variables.

All types of constants (integer, real, and strings) can be read, but not expressions. Items are separated by commas. Strings containing commas, colons or spaces must be put in quotes.

RUN initialises the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is the programmer's responsibility 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.

RESTORE may be used to set the data pointer to a specific line for subsequent readings.

Remarks: It is good programming practice to put large amounts of **DATA** statements at the end of the program, so they don't slow down the search for line numbers after **GOTO**, and other statements with line number targets.

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# channel, record [, byte]

Usage: Positions the read/write pointer of a relative file.

channel number, which was given to a previous call of commands such as **DOPEN**, or **OPEN**.

record target record (1-65535).

byte byte position in record.

RECORD can only be used for files of type **REL**, which are relative files capable of direct access.

RECORD positions the file pointer to the specified record number. If this record number does not exist and there is enough space on the disk which **RECORD** is writing to, the file is expanded to the requested record count by adding empty records. When this occurs, the disk status will give the message RECORD NOT PRESENT, but this is not an error!

After a call of **INPUT#** or **PRINT#**, the file pointer will proceed to the next record position.

Remarks: The Commodore disk drives have a bug in their DOS, which can destroy data by using relative files. A recommended workaround is to use the command **RECORD** twice, before and after the I/O operation.

Example: Using **RECORD**

```
100 DOPENH#2,"DATA BASE",L240 :REM OPEN OR CREATE
110 FOR IX=1 TO 20 :REM WRITE LOOP
120 PRINT#2,"RECORD #";IX :REM WRITE RECORD
130 NEXT IX :REM END LOOP
140 DCLOSE#2 :REM CLOSE FILE
150 :REM NOW TESTING
160 DOPENH#2,"DATA BASE",L240 :REM REOPEN
170 FOR IX=20 TO 2 STEP -2 :REM READ FILE BACKWARDS
180 RECORD#2,IX :REM POSITION TO RECORD
190 INPUT#2,A$ :REM READ RECORD
200 PRINT A$;:IF IX AND 2 THEN PRINT
210 NEXT IX :REM LOOP
220 DCLOSE#2 :REM CLOSE FILE

RUN
RECORD # 20 RECORD # 18
RECORD # 16 RECORD # 14
RECORD # 12 RECORD # 10
RECORD # 8 RECORD # 6
RECORD # 4 RECORD # 2
```

REM

Token: \$8F

Format: **REM**

Usage: Marks any characters after **REM** on the same line as a comment.

Characters after **REM** are never executed, they're ignored by BASIC.

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 brackets, e.g. (F1\$).

new is either a quoted string, e.g. "BACKUP" or a string expression in brackets, e.g. (F9\$).

drive drive # in dual drive disk units.

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

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

Remarks: **RENAME** is executed in the DOS of the disk drive. It can rename all regular file types (**PRG**, **REL**), **SEQ**, **USR**. The old file must exist, and the new file must not exist. Only single files can be renamed, wildcard characters such as '*' 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, range}]

Usage: Used to renumber all, or a range of lines of a BASIC program.

new new starting line of the line range to renumber. The default value is 10.

inc increment to be used. The default value is 10.

range line range to renumber. The default values are from first to last line.

RENUMBER executes in either space conserving mode or optimisation mode. Optimisation mode removes space characters before line numbers, thereby reducing code size and decreasing execution time, while the space conserving leaves spaces untouched. Optimisation mode is triggered by typing the first argument, (the **new** starting number), adjacent to the keyword **RENUMBER** with no space in between.

RENUMBER changes all line numbers in the chosen range and also changes all references in statements that use **GOSUB**, **GOTO**, **RESTORE**, **RUN**, **TRAP**, etc.

RENUMBER can only be executed in direct mode. If it detects a problem such as memory overflow, unresolved references or line number overflow (more than than 64000 lines), it will stop with an error message and leave the program unchanged.

RENUMBER may be called with 0–3 parameters. Unspecified parameters use their default values.

Remarks: **RENUMBER** may need several minutes to execute for large programs.

Examples: Using **RENUMBER**

```
RENUMBER           :REM SPACE CONSERVING, NUMBERS WILL BE 10,20,30,....
RENUMBER 100,5      :REM SPACE CONSERVING, NUMBERS WILL BE 100,105,110,115,...
RENUMBER601,1,500   :REM OPTIMISATION, RENUMBER STARTING AT 500 TO 601,602,...
RENUMBER 100,5,120-180 :REM SPACE CONSERVING RENUMBER LINES 120-180 TO 100,105,....

10 GOTO 20
20 GOTO 10
RENUMBER 100,10      :REM SPACE CONSERVING
100 GOTO 110
110 GOTO 100
RENUMBER100,10      :REM OPTIMISATION
100 GOTO110
110 GOTO100
```

RESTORE

Token: \$8C

Format: RESTORE [line]

Usage: Set, or reset the internal pointer for **READ** from **DATA** statements.

line new position for the pointer. The default is the first program line.

Remarks: The new pointer target **line** does not need to contain **DATA** statements. Every **READ** will advance the pointer to the next **DATA** statement automatically.

Example: Using **RESTORE**

```
10 DATA 3,1,4,1,5,9,2,6
20 DATA "MEGAB5"
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: Used in a **TRAP** routine to resume normal program execution after handling an error.

RESUME with no parameters attempts to re-execute the statement that caused the error. The **TRAP** routine should have examined and corrected the issue where the error occurred.

line line number to resume program execution at.

NEXT resumes execution following the statement that caused the error. This could be the next statement on the same line (separated with a colon ':'), or the statement on the next line.

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 from to call the handler.

Example: Using RETURN

```
10 SCNCLR      :REM CLEAR SCREEN
20 FOR I=1 TO 20 :REM DEFINE LOOP
30 GOSUB 100    :REM CALL SUBROUTINE
40 NEXT I       :REM LOOP
50 END          :REM END OF PROGRAM
100 CURSOR ON,I,I,0 :REM ACTIVATE AND POSITION CURSOR
110 PRINT "X";
120 SLEEP 0.5   :REM WAIT 0.5 SECONDS
130 CURSOR OFF  :REM SWITCH BLINKING CURSOR OFF
140 RETURN       :REM RETURN TO CALLER
```

RGRAPHIC

Token: \$CC

Format: RGRAPHIC(screen, parameter)

Usage: Return graphic screen status and parameters

Parameter	Description
0	Open (1), Closed (0), or Invalid (>1)
1	Width (0=320, 1=640)
2	Height (0=200, 1=400)
3	Depth (1-8 Bitplanes)
4	Bitplanes Used (Bitmask)
5	Bank 4 Blocks Used (Bitmask)
6	Bank 5 Blocks Used (Bitmask)
7	Drawscreen # (0-3)
8	Viewscreen # (0-3)
9	Drawmodes (Bitmask)
10	pattern type (bitmask)

Example: Using RGRAPHIC

```
10 GRAPHIC CLR      :REM INITIALISE
20 SCREEN DEF 0,1,0,4 :REM SCREEN 0:640 X 200 X 4
30 SCREEN OPEN 0      :REM OPEN
40 SCREEN SET 0,0      :REM DRAW = VIEW = 0
50 SCNCLR 0          :REM CLEAR
60 PEN 0,1            :REM SELECT COLOUR
70 LINE 0,0,639,199   :REM DRAW LINE
80 FOR I=0 TO 10:A(I)=RGRAPHIC(0,I) :NEXT
90 SCREEN CLOSE 0
100 FOR I=0 TO 6:PRINT I;A(I):NEXT :REM PRINT INFO
```

RUN

```
0 1
1 1
2 0
3 4
4 15
5 15
6 15
```

RIGHT\$

Token: \$C9

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

Usage: Returns a string containing the last *n* characters from **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** x variable, y variable, button variable

Usage: Reads mouse position and button status.

x variable numeric variable where the x-position will be stored.

y variable numeric variable where the y-position will be stored.

button variable numeric 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

RMOUSE places -1 into all variables if the mouse is not connected or disabled.

Remarks: 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, as the numbers are not *really* random. They are derived from another number called a "seed" that generates reproducible sequences. **type** determines which seed is used:

- **type = 0** use system clock.
- **type < 0** use the value of **type** as seed.
- **type > 0** derive a new random number from previous one.

Remarks: Seeded random number sequences produce the same sequence for identical seeds.

Example: Using RND:

```
10 DEF FD1(X) = INT(RND(0)*6)+1 :REM DICE FUNCTION
20 FOR I=1 TO 10                  :REM THROW 10 TIMES
30 PRINT I;FD1(0)                 :REM PRINT DICE POINTS
40 NEXT
```

RPALETTE

Token: \$CE \$0D

Format: RPALETTE(screen, index, rgb)

Usage: Returns the red, green or blue value of a palette colour index.

screen screen number (0-3).

index palette colour index.

rgb (0: red, 1: green, 2:blue).

Example: Using RPALETTE

```
10 SCREEN 320,200,4 :REM DEFINE AND OPEN SCREEN
20 R = RPALETTE(0,3,0) :REM GET RED
30 G = RPALETTE(0,3,1) :REM GET GREEN
40 B = RPALETTE(0,3,2) :REM GET BLUE
50 SCREEN CLOSE :REM CLOSE SCREEN
60 PRINT "PALETTE INDEX 3 RGB =",R;G;B
```

RUN

PALETTE INDEX 3 RGB = 0 15 15

RPEN

Token: \$D0

Format: RPEN(n)

Usage: Returns the colour index of pen n.

n pen number (0-2), where:

- 0 draw pen.
- 1 erase pen.
- 2 outline pen.

Example: Using RPEN

```
10 GRAPHIC CLR      :REM INITIALISE
20 SCREEN DEF 0,1,0,4 :REM SCREEN 0:640 X 200 X 4
30 SCREEN OPEN 0     :REM OPEN
40 SCREEN SET 0,0    :REM DRAW = VIEW = 0
50 SCNCLR 0          :REM CLEAR
60 PEN 0,1           :REM SELECT COLOUR
70 X = RPEN(0)
80 Y = RPEN(1)
90 C = RPEN(2)
100 SCREEN CLOSE 0
110 PRINT "DRAW PEN COLOUR = ";X
RUN
DRAW PEN COLOUR = 1
```

RPLAY

Token: \$FE \$0F

Format: RPLAY(voice)

Usage: Returns a value of 1 or 0, to indicate whether a melody is playing on the given voice channel or not.

voice the voice channel to assess, ranging from 1 to 6.

Example: Using RPLAY:

```
10 PLAY "04ICDEFGAB05CR","02QCGEGC01GCR"  
30 IF RPLAY(1) OR RPLAY(2) THEN GOTO 30: REM WAIT FOR END OF SONG
```

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 gets accumulator value.

xreg gets X register value.

yreg gets Y register value.

zreg gets Z register value.

sreg gets status register value.

Remarks: The register values after a **SYS** call are stored in system memory. This is how **RREG** is able to retrieve them.

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 sprite multi-colour number:

- **1** get multi-colour # 1.
- **2** get multi-colour # 2.

Remarks: Refer to **SPRITE** and **SPRCOLOR** for more information.

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
```

RSPEED

Token: \$CE \$0E

Format: **RSPEED(n)**

Usage: Returns the current CPU clock in MHz.

n numeric dummy argument, which is ignored.

Remarks: **RSPEED(n)** will not return the correct value if **POKE 0,65** has previously been used to enable the highest speed (40MHz).

Refer to the **SPEED** command for more information.

Example: Using **RSPEED**:

```
10 X=RSPEED(0)      :REM GET CLOCK
20 IF X=1 THEN PRINT "1 MHZ" :GOTO 50
30 IF X=3 THEN PRINT "3.5 MHZ" :GOTO 50
40 IF X=40 THEN PRINT "40 MHZ"
50 END
```

RSPPoS

Token: \$CE \$05

Format: RSPPoS(sprite, n)

Usage: Returns a sprite's position and speed

sprite sprite number.

n sprite parameter to retrieve:

- **0** X position.
- **1** Y position.
- **2** speed.

Remarks: Refer to the **MOVSPR** and **SPRITE** commands for more information.

Example: Using RSPPoS:

```
10 SPRITE 1,1      :REM TURN SPRITE 1 ON
20 XP = RSPPoS(1,0) :REM GET X OF SPRITE 1
30 YP = RSPPoS(1,1) :REM GET Y OF SPRITE 1
30 SP = RSPPoS(1,2) :REM GET SPEED OF SPRITE 1
```

RSprite

Token: \$CE \$06

Format: **RSprite**(sprite, n)

Usage: Returns a sprite's parameter.

sprite sprite number (0-7).

n the sprite parameter to return (0-5):

- **0** turned on (0 or 1) A 0 means the sprite is off.
- **1** foreground colour (0-15).
- **2** background priority (0 or 1).
- **3** x-expanded (0 or 1). 0 means it's not expanded.
- **4** y-expanded (0 or 1). 0 means it's not expanded.
- **5** multi-colour (0 or 1). 0 means it's not multi-colour.

Remarks: Refer to the **MOVSPR** and **SPRITE** commands for more information.

Example: Using **RSprite**:

```
10 SPRITE 1,1      :REM TURN SPRITE 1 ON
20 EN = RSsprite(1,0) :REM SPRITE 1 ENABLED ?
30 FG = RSsprite(1,1) :REM SPRITE 1 FOREGROUND COLOUR INDEX
40 BP = RSsprite(1,2) :REM SPRITE 1 BACKGROUND PRIORITY
50 XE = RSsprite(1,3) :REM SPRITE 1 X EXPANDED ?
60 YE = RSsprite(1,4) :REM SPRITE 1 Y EXPANDED ?
70 MC = RSsprite(1,5) :REM SPRITE 1 MULTI-COLOUR ?
```

RUN

Token: \$8A

Format: **RUN** [line number]

RUN filename [,D drive] [,U unit] ↑ filename

Usage: (Load and) run a BASIC program.

If a filename is given, the program file is loaded into memory and run, otherwise the program that is currently in memory will be used instead.

The ↑ can be used as shortcut, if used in direct mode at the leftmost column. It can be used to load and run a program from a dir listing by moving the cursor to the row with the filename, typing the ↑ at the start of the row and pressing return. Characters before and after the quoted filename, will be ignored (like the PRG for example).

line number an existing line number of the program in memory to run from.

filename either a quoted string, e.g. "PROG" or a string expression in brackets, 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 such as the 1541, 1571, or 1581.

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

RUN first resets all internal pointers to their default values. Therefore, there will be no variables, arrays or strings defined. The run-time stack is also reset, and the table of open files is cleared.

Remarks: To start or continue program execution without resetting everything, use **GOTO** instead.

Examples: Using **RUN**

```
RUN "FLIGHTSIM" :REM LOAD AND RUN PROGRAM FLIGHTSIM  
RUN 1000      :REM RUN PROGRAM IN MEMORY, START AT LINE# 1000  
RUN          :REM RUN PROGRAM IN MEMORY
```

RWINDOW

Token: \$CE \$09

Format: **RWINDOW(n)**

Usage: Returns information regarding the current text window.

n the screen parameter to retrieve:

- 0 width of current text window.
- 1 height of current text window.
- 2 number of columns on screen (40 or 80).

Remarks: Older versions of **RWINDOW** reported the width - 1 and the height - 1 for arguments 0 and 1.

Refer to the **WINDOW** command for more information.

Example: Using **RWINDOW**:

```
10 W = RWINDOW(2)      :REM GET SCREEN WIDTH
20 IF W=80 THEN BEGIN   :REM IS 80 COLUMNS MODE ACTIVE?
30   PRINT CHR$(27)+"X"; :REM YES, SWITCH TO 40COLUMNS
40 BEND
```

SAVE

Token: \$94

Format: **SAVE** filename [, unit]

 filename [, unit]

Usage: Saves a BASIC program to a file of type **PRG**.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

The maximum length of the filename is 16 characters, not counting the optional save and replace character 'e' 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 not recommended to use this option on 1541 and 1571 drives, as they contain a "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 from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **SAVE** is obsolete, implemented only for backwards compatibility. **DSAVE** should be used instead. The shortcut symbol  is next to **1**. Can only be used in direct mode.

Examples: Using **SAVE**

```
SAVE "ADVENTURE"  
SAVE "ZORK-I",8  
SAVE "1:DUNGEON",9
```

SAVEIFF

Token: \$FE \$44

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

Usage: Saves a picture from memory to a disk file in **IFF** format. The IFF (Interchange File Format) is supported by many different applications and operating systems. **SAVEIFF** saves the image, the palette and resolution parameters.

filename is either a quoted string such as "**DATA**", or a string expression in brackets such as **(I\$)**. 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 not recommended to use this option on 1541 and 1571 drives, as they contain a "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 such as the 1541, 1571, or 1581.

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

Remarks: Files saved with **SAVEIFF** can be loaded with **LOADIFF**. Tools are available to convert popular image formats to IFF. These tools are available on several operating systems, such as AMIGA OS, macOS, Linux, and Windows. For example, **ImageMagick** is a free graphics package that includes a tool called **convert**, which can be used to create IFF files in conjunction with the **ppmtoilbm** tool from the **Netpbm** package.

Example: Using **SAVEIFF**

```
10 SCREEN 320,200,2      :REM SCREEN #0 320 X 200 X 2
20 PEN 1                  :REM DRAWING PEN COLOR 1 (WHITE)
30 LINE 25,25,295,175    :REM DRAW LINE
40 SAVEIFF "LINE-EXAMPLE",U8 :REM SAVE CURRENT VIEW TO FILE
50 SCREEN CLOSE            :REM CLOSE SCREEN AND RESTORE PALETTE
```

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 given colour.

Example: Using **SCNCLR**:

```
1 REM SCREEN EXAMPLE 2
10 GRAPHIC CLR      :REM INITIALIZE
20 SCREEN DEF 1,0,0,2 :REM SCREEN #1 320 X 200 X 2
30 SCREEN OPEN 1     :REM OPEN SCREEN 1
40 SCREEN SET 1,1    :REM USE SCREEN 1 FOR RENDERING AND VIEWING
50 SCREEN CLR 0      :REM CLEAR SCREEN
60 PALETTE 1,1,15,15,15 :REM DEFINE COLOUR 1 AS WHITE
70 PEN 0,1           :REM DRAWING PEN
80 LINE 25,25,295,175 :REM DRAW LINE
90 SLEEP 10          :REM WAIT FOR 10 SECONDS
100 SCREEN CLOSE 1   :REM CLOSE SCREEN AND RESTORE PALETTE
```

SCRATCH

Token: \$F2

Format: SCRATCH filename [,D drive] [,U unit] [,R]

Usage: Used to erase a disk file.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

drive drive # in dual drive disk units.

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

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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 disk.

Remarks: SCRATCH filename is a synonym of ERASE filename and DELETE filename.

In direct mode the success and the number of erased files is printed. The second to last number from the message contains the number of successfully erased files,

Examples: Using SCRATCH

```
SCRATCH "DRM",U9 :REM ERASE FILE DRM ON UNIT 9  
81, FILES SCRATCHED,01,00  
SCRATCH "OLD%" :REM ERASE ALL FILES BEGINNING WITH "OLD"  
01, FILES SCRATCHED,04,00  
SCRATCH "R*=PRG" :REM ERASE PROGRAM FILES STARTING WITH 'R'  
01, FILES SCRATCHED,09,00
```

SCREEN

Token: \$FE \$2E

Format:

- SCREEN** [screen,] width, height, depth
- SCREEN CLR** colour
- SCREEN DEF** width flag, height flag, depth
- SCREEN SET** drawscreen, viewscreen
- SCREEN OPEN** [screen]
- SCREEN CLOSE** [screen]

Usage: There are two approaches available when preparing the screen for the drawing of graphics: a simplified approach, and a detailed approach.

Simplified approach:

The first version of **SCREEN** (which has pixel units for width and height) is the easiest way to start a graphics screen, and is the preferred method if only a single screen is needed (i.e., a second screen isn't needed for double buffering). This does all of the preparatory work for you, and will call commands such as **GRAPHIC CLR**, **SCREEN CLR**, **SCREEN DEF**, **SCREEN OPEN** and, **SCREEN SET** on your behalf. It takes the following parameters:

SCREEN [screen,] width, height, depth

- **screen** the screen number (0-3) is optional. If no screen number is given, screen 0 is used. To keep this approach as simple as possible, it is suggested to use the default screen 0.
- **width** 320 or 640 (default 320)
- **height** 200 or 400 (default = 200)
- **depth** 1..8 (default = 8), colours = 2^{depth} .

The argument parser is error tolerant and uses default values for width (320) and height (200) if the parsed argument is not valid.

This version of **SCREEN** starts with a predefined palette and sets the background to black, and the pen to white, so drawing can start immediately using the default values.

On the other hand, the detailed approach will require the setting of palette colours and pen colour before any drawing can be done.

The **colour** value must be in the range of 0 to 15. Refer to the colour table under **BACKGROUND** on page [B-25](#) for the **colour** values and their corresponding colours.

When you are finished with your graphics screen, simply call **SCREEN CLOSE** [screen] to return to the text screen.

Detailed approach:

The other versions of **SCREEN** perform special actions, used for advanced graphics programs that open multiple screens, or require "double buffering". If you have chosen the simplified approach, you will not require any of these versions below, apart from **SCREEN CLOSE**.

SCREEN CLR colour (or **SCNCLR colour**)

Clears the active graphics screen by filling it with **colour**.

SCREEN DEF screen, width flag, height flag, depth

Defines resolution parameters for the chosen screen. The width flag and height flag indicate whether high resolution (1) or low resolution (0) is chosen.

- **screen** screen number 0-3
- **width flag** 0-1 (0:320, 1:640 pixel)
- **height flag** 0-1 (0:200, 1:400 pixel)
- **depth** 1-8 (2 - 256 colours)

Note that the width and height values here are **flags**, and **not pixel units**.

SCREEN SET drawscreen, viewscreen

Sets screen numbers (0-3) for the drawing and the viewing screen, i.e., while one screen is being viewed, you can draw on a separate screen and then later flip between them. This is what's known as double buffering.

SCREEN OPEN screen

Allocates resources and initialises the graphics context for the selected **screen** (0-3). An optional variable name as a further argument, gets the result of the command that can be tested afterwards for success.

SCREEN CLOSE [screen]

Closes **screen** (0-3) and frees resources. If no value is given, it will default to 0. Also note that upon closing a screen, **PALETTE RESTORE** is automatically performed for you.

Examples: Using **SCREEN**:

```
5 REM *** SIMPLIFIED APPROACH ***
10 SCREEN 320,200,2 :REM SCREEN #0: 320 X 200 X 2
20 PEN 1 :REM DRAWING PEN COLOUR = 1 (WHITE)
30 LINE 25,25,295,175 :REM DRAW LINE
40 GETKEY A$ :REM WAIT KEYPRESS
50 SCREEN CLOSE :REM CLOSE SCREEN 0 (RESTORE PALETTE)
```

```
5 REM *** DETAILED APPROACH ***
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 PALETTE 1,1,15,15,15:REM DEFINE COLOUR 1 AS WHITE
70 PEN 0,1 :REM DRAWING PEN
80 LINE 25,25,295,175 :REM DRAW LINE
90 SLEEP 10 :REM WAIT 10 SECONDS
100 SCREEN CLOSE 1 :REM CLOSE SCREEN 1 (RESTORE PALETTE)
```

SET

- Token:** \$FE \$2D
- Format:** **SET DEF** unit
SET DISK old **TO** new
SET VERIFY <ON | OFF>
- Usage:** **SET DEF** 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 is used to change the unit number of a disk drive temporarily.
SET VERIFY enables or disables the DOS verify-after-write mode for 3.5 drives.
- Remarks:** These settings are valid until a reset or shutdown.
- Examples:** 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)
SET VERIFY ON  :REM ACTIVATE VERIFY-AFTER-WRITE MODE
```

SETBIT

Token: \$FE \$2D \$FE \$4E

Format: **SETBIT** address, bit number

Usage: Sets a single bit at the **address**.

If the address is in the range of \$0000 to \$FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to \$10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

The **bit number** is a value in the range of 0-7.

A bank value > 127 is used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **SETBIT**

```
10 BANK 128      :REM SELECT SYSTEM MAPPING
20 SETBIT $D011,6 :REM ENABLE EXTENDED BACKGROUND MODE
30 SETBIT $D01B,0 :REM SET BACKGROUND PRIORITY FOR SPRITE 0
```

SGN

Token: \$B4

Format: **SGN**(numeric expression)

Usage: Extracts the sign from the argument and returns it as a number:

- **-1** negative argument.
- **-0** zero.
- **1** 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: \$BF

Format: **SIN**(numeric expression)

Usage: Returns the sine of the **numeric expression**. 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 multiplying it with $\pi/180$.

Examples: Using **SIN**

```
PRINT SIN(0.7)
.644217687

X=30:PRINT SIN(X * pi / 180)
.5
```

SLEEP

Token: \$FE \$0B

Format: **SLEEP** seconds

Usage: Pauses execution for the given duration. The argument is a positive floating point number. The precision is 1 microsecond.

Remarks: Pressing **RUN STOP** interrupts the sleep.

Example: Using **SLEEP**

```
20 SLEEP 10      :REM WAIT 10 SECONDS
40 SLEEP 0.0005 :REM SLEEP 500 MICRO SECONDS
50 SLEEP 0.01    :REM SLEEP 10 MILLI SECONDS
60 SLEEP DD      :REM TAKE SLEEP TIME FROM VARIABLE DD
70 SLEEP 600     :REM SLEEP 10 MINUTES
```

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 in jiffies (0-32767). The duration of a jiffy depends on the display standard. There are 50 jiffies per second with PAL, 60 per second with NTSC.

dir direction (0:up, 1:down, 2:oscillate).

min minimum frequency (0-65535).

sweep sweep range (0-65535).

wave waveform (0:triangle, 1:sawtooth, 2:square, 3:noise).

pulse pulse width (0-4095).

Remarks: **SOUND** starts playing the sound effect and immediately continues with the execution of the next BASIC statement while the sound effect is played. This enables the showing of graphics or text and playing sounds simultaneously.

Note that **SOUND** makes use of SID2 (for voices 1 to 3) and SID4 (for voices 4 to 6) of the 4 SID chips of the system. Also note that, by default, SID1 and SID2 are slightly right-biased and SID3 and SID4 are slightly left-biased (in terms of stereo sound).

The 6 voice channels used by the **SOUND** command (on SID2+SID4) are distinct to the 6 channels used by the **PLAY** command (on SID1+SID3).

Examples: Using **SOUND**

```
IF PEEK($D06F) AND $80 THEN J = 60: ELSE J = 50 :REM J IS JIFFIES PER SECOND
SOUND 1, 7382, J :REM PLAY SQUARE WAVE ON VOICE 1 FOR 1 SECOND
SOUND 2, 800, J*60 :REM PLAY SQUARE WAVE ON VOICE 2 FOR 1 MINUTE
SOUND 3, 4000, 120, 2, 2000, 400, 1 :REM PLAY SWEEEPING SAWTOOTH WAVE ON VOICE 3
```

SPC

Token: \$A6

Format: **SPC**(columns)

Usage: Skips **columns**.

The effect is similar to pressing → <column> times.

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
```

SPEED

Token: \$FE \$26

Format: **SPEED** [speed]

Usage: Set CPU clock to 1MHz, 3.5MHz or 40MHz.

speed CPU clock speed where:

- **1** sets CPU to 1MHz.
- **3** sets CPU to 3MHz.
- Anything other than **1** or **3** sets the CPU to 40MHz.

Remarks: Although it's possible to call **SPEED** with any real number, the precision part (the decimal point and any digits after it), will be ignored.

SPEED is a synonym of **FAST**.

SPEED has no effect if **POKE 0,65** has previously been used to set the CPU to 40MHz.

Example: Using **SPEED**

```
10 SPEED      :REM SET SPEED TO MAXIMUM (40 MHZ)
20 SPEED 1    :REM SET SPEED TO 1 MHZ
30 SPEED 3    :REM SET SPEED TO 3.5 MHZ
40 SPEED 3.5  :REM SET SPEED TO 3.5 MHZ
```

SPRCOLOR

Token: \$FE \$08

Format: **SPRCOLOR** [{mc1, mc2}]

Usage: Sets multi-colour sprite colours.

Sprite, which sets the attributes of a sprite, only sets the foreground colour. For setting the additional two colours of multi-colour sprites, use **SPRCOLOR** instead.

Remarks: The colours used with **SPRCOLOR** will affect all sprites. Refer to the **Sprite** command for more information.

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 CLR**

SPRITE LOAD filename [,D drive] [,U unit]

SPRITE SAVE filename [,D drive] [,U unit]

SPRITE num [{, switch, colour, prio, expx, expy, mode}]

Usage: **SPRITE CLR** clears all sprite data and sets all pointers and attributes to their default values.

SPRITE LOAD loads sprite data from **filename** to sprite memory.

SPRITE SAVE saves sprite data from sprite memory to **filename**.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

The last form switches a sprite on or off and sets its attributes:

num 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: **SPRCOLOR** must be used to set additional colours for multi-colour sprites (mode = 1).

Example: Using **SPRITE**:

```
2290 CLR:SCNCLR:SPRITE CLR
2300 SPRITE LOAD "DEMOSPRITES1"
2320 FORI=0TO7: C=I: IFC=6THENC=8
2330 MOVSPr I, 60+30*I,0 TO 60+30*I,65+20*I, 3:SPRITE I,I,C,,1,1:NEXT: SLEEP3
2340 FORI=0TO7: SPRITE I,,,0,0 :NEXT: SLEEP3: SPRITE CLR
2350 FORI=0TO7: MOVSPr I,45*I#5 :NEXT: FORI=0TO7: SPRITE I,I: NEXT
2360 FORI=0TO7:X=60+30*I:Y=65+20*I:DO
2370 LOOPUNTIL(X=RSPPOS(I,,))AND(Y=RSPPOS(I,1)):MOVSPr,,#:NEXT
```

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: Source and destination can either be a sprite number or a string variable,

SPRSAV can be used with the basic form of sprites (C64 compatible) only. These sprites occupy 64 bytes of memory, and create strings of length 64, if the destination parameter is a string variable.

Extended sprites and variable height sprites cannot be used with **SPRSAV**.

A string array of sprite data can be used to store many shapes and copy them fast to the sprite memory with the command **SPRSAV**.

It's also a convenient method to read or write shapes of single sprites from or to a disk file.

Example: Using **SPRSAV**:

```
10 SPRITE LOAD "SPRITEDATA" :REM LOAD DATA FOR 8 SPRITES
20 SPRITE 1,1 :REM TURN SPRITE 1 ON
30 SPRSAV 1,2 :REM COPY SPRITE 1 DATA TO SPRITE 2
40 SPRITE 2,1 :REM TURN SPRITE 2 ON
50 SPRSAV 1,A$ :REM SAVE SPRITE 1 DATA IN STRING A$
```

SQR

Token: \$BA

Format: **SQR**(numeric expression)

Usage: Returns the square root of the numeric expression.

Remarks: The argument must not be negative.

Example: Using **SQR**

```
PRINT SQR(2)
```

```
1.41421356
```

ST

Format: ST

Usage: ST holds the status of the last I/O operation. If ST is zero, there was no error, otherwise it is set to a device dependent error code.

Remarks: ST is a reserved system variable.

Example: Using ST

```
100 MX=100:DIM T$(MX)      :REM DATA ARRAY
110 DOPEN#1,"DATA"          :REM OPEN FILE
120 IF DS THEN PRINT"COULD NOT OPEN":STOP
130 LINE INPUT#1,T$(N):N=N+1 :REM READ ONE RECORD
140 IF N>MX THEN PRINT "TOO MANY DATA":GOTO 160
150 IF ST=0 THEN 130        :REM ST = 64 FOR END-OF-FILE
160 DCLOSE#1
170 PRINT "READ";N;" RECORDS"
```

STEP

Token: \$A9

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

Usage: **STEP** is an optional part of a **FOR** loop.

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

start initial value of the index.

end is checked at the end of an iteration, and determines whether another iteration will be performed, or if the loop will exit.

step defines the change applied to the **index** at the end of a loop iteration. 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 than or equal to **start**. For negative increments, **end** must be less than or equal to **start**.

It is bad programming practice to change the value of the **index** variable inside the loop or to jump into or out of a 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 will be displayed showing the line number where the program stopped. The **READY**. prompt appears and the computer goes into direct mode, waiting for keyboard input.

Remarks: All variable definitions are still valid after **STOP**. They may be inspected or altered, and the program may be continued with **CONT**. However, any editing of the program source will disallow any further continuation.

Program execution can be resumed with **CONT**.

Example: Using **STOP**

```
10 IF V < 0 THEN STOP : REM NEGATIVE NUMBERS STOP THE PROGRAM  
20 PRINT SQR(V)      : REM PRINT SQUARE ROOT
```

STR\$

Token: \$C4

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

Usage: Returns a string containing the formatted value of the argument, as if it were **PRINTed** to the string.

Example: Using **STR\$**:

```
A$ = "THE VALUE OF PI IS " + STR$(π)  
PRINT A$  
THE VALUE OF PI IS 3.14159265
```

SYS

Token: \$9E

Format: **SYS** address [{, areg, xreg, yreg, zreg, sreg}]

Usage: Calls a machine language subroutine. This can be a ROM-resident kernal routine or any other routine which has previously been loaded or **POKEd** to RAM.

The CPU registers are loaded with the arguments (if they're specified), then a subroutine call (**JSR address**) is performed. **JSR** is an assembly language instruction that is short for **Jump to SubRoutine**. The called routine should exit with an **RTS** instruction. **RTS** is another assembly language instruction that is short for **Return from Subroutine**. After the subroutine has returned, the register contents will be saved, and the execution of the BASIC program will continue.

address start address of the subroutine.

If the address value is 16 bit (\$0000 - \$FFFF), the bank value, that is currently valid (see **BANK**) is examined. A bank value of 128 lets the current mapping persist. That is: RAM is only available at the address range (\$0000 - \$1FFF), while BASIC ROM, KERNAL and I/O occupy the rest. Short machine language programs may use the address range (\$1800 - \$1FFF) which is only used by BASIC while a graphics screen is open.

If the address is higher than \$FFFF, it is interpreted as a linear 24 bit address and the value of **BANK** is ignored. The initial mapping is shown in the following table:

Range	Content
0000 - 1FFF	bank 0 with direct page, stack, vectors and interface routines
2000 - BFFF	selected RAM bank:address bits 16-23
C000 - CFFF	kernal ROM
D000 - DFFF	I/O
E000 - EFFF	editor ROM
F000 - FFFF	kernal ROM and jump table

The RAM banks 0, 1, 4 and 5 may be used on a MEGA65 with the **SYS** command. The attic RAM cannot be used for this purpose, because the 24 bit address of the **SYS command** is limited to the lower 16MB of the address range.

areg CPU accumulator value.

xreg CPU X register value.

yreg CPU Y register value.

zreg CPU Z register value.

sreg Status register value.

Remarks: The register values after a **SYS** call are stored in system memory. **RREG** can be used to retrieve these values.

The **SYS** instruction on the MEGA65 is completely different to the well known **SYS** command on the C64. It is not possible to have the BASIC ROM and a BASIC program, in the same mapping because they occupy the same address range.

Using **SYS** properly (i.e. without corrupting the system), requires some technical skill, which is out of scope of the User's Guide. However, if you would like to learn more, there is a lot more information and examples in the MEGA65 Developer's Guide.

Example: Using **SYS**:

```
10 REM DEMO FOR SYS:CHANGING THE BORDER COLOUR
20 BANK 0
30 POKE $4000,$EE,$20,$00,$60 :REM INC $0020:RTS
40 SYS $4000           :REM CALL SUBROUTINE AT $4000 / BANK $00
50 GETKEY A$:IF A$ <> "Q" THEN 40
```

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 the cursor will not move. The column count starts with 0 being the left-most column.

Remarks: This function shouldn't be confused with **TAB**, 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. The result is in the range (-1.0 to +1.0)

Remarks: An argument in units of degrees can be converted to radians by multiplying it 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 PLAY.

speed 1-255.

The duration (in seconds) of a whole note is computed with *duration* = $24/speed$.

Example: Using TEMPO

```
10 VOL 8
20 FOR T = 24 TO 18 STEP -2
30 TEMPO T
40 PLAY "T0M3040GAGFED","T204M5P0H.DP56B","T503IGAGAGAABABAB"
50 IF RPLAY(1) THEN GOTO 50
60 NEXT T
70 PLAY "T0050C04GEH.C","T205IEFEDEDCE606P8CP0R","T503ICDCDEFEDC04C"
```

THEN

Token: \$A7

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

Usage: **THEN** is part of an **IF** statement.

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

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

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

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

Example: Using **THEN**

```
1 REM THEN
10 RED$=CHR$(28) : BLACK$=CHR$(144) : WHITE$=CHR$(5)
20 INPUT "ENTER A NUMBER";V
30 IF V<0 THEN PRINT RED$; : ELSE PRINT BLACK$;
40 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
50 PRINT WHITE$
60 INPUT "END PROGRAM: (Y/N)"; A$
70 IF A$="Y" THEN END
80 IF A$="N" THEN 20 : ELSE 60
```

TI

Format: TI

Usage: TI is a high precision timer with a resolution of 1 micro second.

It is started or reset with **CLR TI**, and can be accessed in the same way as any other variable in expressions.

Remarks: TI is a reserved system variable. The value in TI is the number of seconds (to 6 decimal places) since it was last cleared or started.

Example: Using TI

```
100 CLR TI           :REM START TIMER
110 FOR I% = 1 TO 10000:NEXT :REM DO SOMETHING
120 ET = TI           :REM STORE ELAPSED TIME IN ET
130 PRINT "EXECUTION TIME:";ET;" SECONDS"
```

TI\$

Format: TI\$

Usage: TI\$ stores the time information of the RTC (Real-Time Clock) in text form, using the format: "hh:mm:ss". It is updated with every use.

TI\$ is a read-only variable, which reads the registers of the RTC and formats the values to a string.

Remarks: TI\$ is a reserved system variable.

It is possible to access the RTC registers directly via PEEK. The start address of the registers is at \$FFD7110.

For more information on how to set the Real-Time Clock, refer to the Configuration Utility section on page [4-12](#).

```
100 REM ***** READ RTC ***** ALL VALUES ARE BCD ENCODED
110 RT = $FFD7110          :REM ADDRESS OF RTC
120 FOR I=0 TO 5          :REM SS,MM,HH,DD,MO,YY
130 T(I)=PEEK(RT+I)       :REM READ REGISTERS
140 NEXT I                :REM USE ONLY LAST TWO DIGITS
150 T(2) = T(2) AND 127    :REM REMOVE 24H MODE FLAG
160 T(5) = T(5) + $2000    :REM ADD YEAR 2000
170 FOR I=2 TO 0 STEP -1   :REM TIME INFO
180 PRINT USING ">##";HEX$(T(I));
190 NEXT I
RUN
12 52 36
```

Example: Using TI\$

```
PRINT DT$,TI$
05-APR-2021 15:10:00
```

TO

Token: \$A4

Format: keyword TO

Usage: TO is a secondary keyword used in combination with primary keywords, such as **BACKUP**, **BSAVE**, **CHANGE**, **CONCAT**, **COPY**, **FOR**, **GO**, **RENAME**, and **SET DISK**

Remarks: 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 registers the BASIC error handler. When a program has an error handler, the run-time behaviour changes. Normally, BASIC will exit the program and display an error message.

However, if a BASIC error handler has been registered, BASIC will instead save the execution pointer and line number, place the error number into the system variable ER, and GOTO the line number of TRAP. The trapping routine can examine ER and process the error. From this, the TRAP error handler can then decide whether to STOP or RESUME execution.

TRAP with no argument disables the error handler, and errors will then be handled by the normal system routines.

Example: Using TRAP

```
10 TRAP 100
20 FOR I=1 TO 100
30 PRINT EXP(I)
40 NEXT
50 PRINT "STOPPED FOR I =";I
60 END
100 PRINT ERR$(ER): RESUME 50
```

TROFF

Token: \$D9

Format: TROFF

Usage: Turns off trace mode (switched on by TRON).

Example: Using TROFF

```
10 TRON          :REM ACTIVATE TRACE MODE
20 FOR I=85 TO 100
30 PRINT I;EXP(I)
40 NEXT
50 TROFF         :REM DEACTIVATE TRACE MODE

RUN
[10][20][30] 85  8.22301268E+36
[40][30] 86  2.2352466E+37
[40][30] 87  6.0760302E+37
[40][30] 88  1.65163625E+38
[40][30] 89
?OVERFLOW ERROR IN 30
READY.
```

TRON

Token: \$D8

Format: TRON

Usage: Turns on trace mode.

Example: Using **TRON**

```
10 TRON          :REM ACTIVATE TRACE MODE
20 FOR I=85 TO 100
30 PRINT I;EXP(I)
40 NEXT
50 TROFF         :REM DEACTIVATE TRACE MODE

RUN
[10][20][30] 85  8.22301268E+36
[40][30] 86  2.2352466E+37
[40][30] 87  6.0760302E+37
[40][30] 88  1.65163625E+38
[40][30] 89
?OVERFLOW ERROR IN 30
READY.
```

TYPE

Token: \$FE \$27

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

Usage: Prints the contents of a file containing text encoded as PETSCII.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

drive drive # in dual drive disk units.

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

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

Remarks: **TYPE** cannot be used to print BASIC programs. Use **LIST** for programs instead. **TYPE** can only process **SEQ** or **USR** files containing records of PETSCII text, delimited by the CR character.

The CR character is also known as carriage return, and can be created by using **CHR\$(13)**.

Example: Using **TYPE**

```
TYPE "README"  
TYPE "README 1ST",U$
```

UNLOCK

Token: \$FE \$4F

Format: **UNLOCK** filename/pattern [,D drive] [,U unit]

Usage: Used to unlock files. The specified file or a set of files, that matches the pattern, is unlocked and no more protected. It can be deleted afterwards with the commands **DELETE**, **ERASE** or **SCRATCH**

The command **LOCK** applies the lock.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI\$).

drive drive # in dual drive disk units.

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

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

Remarks: Unlocking a file, that is already unlocked, has no effect.

In direct mode the number of unlocked files is printed. The second to last number from the message contains the number of unlocked files,

Examples: Using **UNLOCK**

```
UNLOCK "SNOOPY",U9 :REM UNLOCK FILE SNOOPY ON UNIT 9  
03,FILES UNLOCKED,01,00  
UNLOCK "BS*"    :REM UNLOCK ALL FILES BEGINNING WITH "BS"  
03,FILES UNLOCKED,04,00
```

UNTIL

Token: \$FC

Format: **DO ... LOOP**

DO [<UNTIL | WHILE> logical expression]
. . . statements [**EXIT**]
LOOP [<UNTIL | WHILE> logical expression]

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

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

Examples: 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; argument

Usage: Parses the **format** string and evaluates the argument. The argument can be either a string or a numeric value. The format of the resulting output is directed by the **format** string.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**. If no channel is specified, the output goes to the screen.

format string variable or a string constant which defines the rules for formatting. When using a number as the **argument**, formatting can be done in either CBM style, providing a pattern such as ####.## or in C style using a <width.precision> specifier, such as %3D %7.2F %4X .

argument the number to be formatted. If the argument does not fit into the format e.g. trying to print a 4 digit variable into a series of three hashes (###), asterisks will be used instead.

Remarks: The format string is only applied for one argument, but it is possible to append more than one **USING format;argument** sequences.

argument 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. If the first character of the format string is an equals '=' sign, the argument string is centered. If the first character of the format string is a greater than '>' sign, the argument string is right justified.

Example: **USING** with a corresponding **PRINT#**

```
PRINT USING "####.###";#, USING " [%.4F] ";SQR(2)
3.14 [1.4142]

PRINT USING " < # # # > ";12*31
< 3 7 2 >

PRINT USING "#####"; "ABCDE"
ABC

PRINT USING ">#####"; "ABCDE"
CDE

PRINT USING "ADDRESS:$%4X";65000
ADDRESS:$FDE8

A$="#####,#####.##":PRINT USING A$;1E8/3
33,333,333.3
```

USR

Token: \$B7

Format: **USR**(numeric expression)

Usage: Invokes an assembly language routine whose memory address is stored at \$02F8 - \$02F9. The result of the **numeric expression** is written to floating point accumulator 1.

After executing the assembly routine, BASIC returns the contents of the floating point accumulator 1.

Remarks: Banks 0-127 give access to RAM or ROM banks. Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

If you would like to learn more, there is a lot more information and examples in the MEGA65 Developer's Guide.

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(r)          :REM PRINT RESULT FOR ARGUMENT PI
```

VAL

Token: \$C5

Format: **VAL**(string expression)

Usage: Converts a string to a floating point value.

This function acts in the same way as reading from a string.

Remarks: A string containing an invalid number will not produce an error, but return 0 as the result instead.

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: **VERIFY** with no **binflag** compares a BASIC program in memory with a disk file of type **PRG**. It does the same as **DVERIFY**, but the syntax is different.

VERIFY with **binflag** compares a binary file in memory with a disk file of type **PRG**. It does the same as **BVERIFY**, but the syntax is different.

filename is either a quoted string, e.g. "PROG" or a string expression.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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. **VERIFY** exits with either the message **OK** or with **VERIFY ERROR**.

VERIFY is obsolete in BASIC 65. It is only here for backwards compatibility. It is recommended to use **DVERIFY** and **BVERIFY** instead.

Examples: Using **VERIFY**

```
VERIFY "ADVENTURE"  
VERIFY "ZORK-I",9  
VERIFY "1:DUNGEON",10
```

VIEWPORT

Token: \$FE \$31

Format: **VIEWPORT CLR**

VIEWPORT DEF x, y, width, height

Usage: **VIEWPORT DEF** defines a clipping region with the origin (upper left position) set to **x, y** and the **width** and **height**. All following graphics commands are limited to the **VIEWPORT** region.

VIEWPORT CLR fills the clipping region with the color of the drawing pen.

Remarks: The clipping region can be reset to full screen by the command **VIEWPORT DEF 0,0,WIDTH,HEIGHT** using the same values for WIDTH and HEIGHT as in the **SCREEN** command.

Example: Using **VIEWPORT**

```
10 SCREEN 320,200,2
20 VIEWPORT DEF 20,30,100,120 :REM REGION 20->119, 30->149
30 PEN 1 :REM SELECT COLOUR 1
40 VIEWPORT CLR :REM FILL REGION WITH COLOUR OF PEN
50 GETKEY A$ :REM WAIT FOR KEYPRESS
60 SCREEN CLOSE
```

VOL

Token: \$DB

Format: VOL volume

Usage: Sets the volume for sound output with **SOUND** or **PLAY**.

volume 0 (off) to 15 (loudest).

Remarks: This volume setting affects all voices.

Example: Using VOL

```
10 TEMPO 22
20 FOR V = 2 TO 8 STEP 2
30 VOL V
40 PLAY "T0M304QGAGFED","T204M5P0H.DP5GB","T503IGAGAGAABABAB"
50 IF RPLAY(1) THEN GOTO 50
60 NEXT V
70 PLAY "T005QCD04GEH.C","T205IEFEDEDCEG06P9CP0R","T503ICDCDEFEDC04C"
```

VSYNC

Token: \$FE \$54

Format: **VSYNC** raster line

Usage: Waits until the selected raster line is active.

raster line (0 - 311) for PAL, (0 - 262) for NTSC mode.

Example: Using **VSYNC**

```
10 IF FRE(-1)<920364 THEN PRINT"UPDATE ROM":END
20 BORDER 3    :REM CHANGE BORDER COLOUR TO CYAN
30 VSYNC 100   :REM WAIT UNTIL RASTER LINE 100
40 BORDER 7    :REM CHANGE BORDER COLOUR TO YELLOW
50 VSYNC 260   :REM WAIT UNTIL RASTER LINE 260
60 GOTO 20    :REM LOOP
```

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 non-zero, otherwise reading is repeated.
This may hang the computer indefinitely if the condition is never met.

Remarks: **WAIT** 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 specific raster line is about to be drawn to the screen.

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 expression]
. . . statements [**EXIT**]
LOOP [<UNTIL | WHILE> logical expression]

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

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

Examples: 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

Remarks: The row values range from 0 to 24. The column values range from 0 to either 39 or 79. This depends on the screen mode.

There can be only one window on the screen. Pressing  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
```

WPEEK

Token: \$CE \$10

Format: **WPEEK**(address)

Usage: Returns an unsigned 16-bit value (word) read from **address** (low byte) and **address+1** (high byte).

If the address is in the range of \$0000 to \$FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to \$10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

Remarks: Banks 0-127 give access to RAM or ROM banks. Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **WPEEK**

```
20 UA = WPEEK($02F8) :REM USR JUMP TARGET  
50 PRINT "USR FUNCTION CALL ADDRESS";UA
```

WPOKE

Token: \$FE \$1D

Format: **WPOKE** address, word [, word ...]

Usage: Writes one or more words into memory or memory mapped I/O, starting at **address**.

If the address is in the range of \$0000 to \$FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to \$10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

word a value from 0-65535. The first word is stored at address (low byte) and address+1 (high byte). The second word is stored at address+2 (low byte) and address+3 (high byte), etc.

Remarks: The address is increased by two for each data word, so a memory range can be written to with a single **WPOKE**.

Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **WPOKE**

```
10 BANK 128      :REM SELECT SYSTEM BANK
20 WPOKE $02F8,$1800  :REM SET USR VECTOR TO $1800
```

XOR

Token: \$E9

Format: operand **XOR** operand

Usage: The Boolean **XOR** operator performs a bit-wise logical 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 (losing precision). Logical operands are converted to 16-bit integer using \$FFFF, (decimal -1) for TRUE, and \$0000 (decimal 0) for FALSE.

Expression	Result
0 XOR 0	0
0 XOR 1	1
1 XOR 0	1
1 XOR 1	0

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

Example: Using **XOR**

```
FOR I = 0 TO 8: PRINT I XOR 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\$

In BASIC, `PRINT CHR$(X)` can be used to print a character from a PETSCII code. Below is the full table of PETSCII codes you can print by index. For example, while in the default uppercase/graphics mode, by using index 65 from the table below as: `PRINT CHR$(65)` you will print the letter `A`. You can read more about **CHR\$** on page [B-47](#).

You can also do the reverse with the `ASC` statement. For example: `PRINT ASC("A")` will output 65, which matches the code in the table.

Note: Function key (F1-F14 + HELP) values in this table are not intended to be printed via `CHR$(X)`, but rather to allow function-key input to be assessed in BASIC programs via the `GET` / `GETKEY` commands.

0	18 	37 %	57 9
1	19 	38 &	58 :
2 UNDERLINE ON	20 	39 '	59 ;
3	21 F10 / BACK WORD	40 (60 <
4	22 F11	41)	61 =
5 WHITE	23 F12 / NEXT WORD	42 *	62 >
6	24 SET/CLEAR TAB	43 +	63 ?
7 BELL	25 F13	44 ,	64 @
8	26 F14 / BACK TAB	45 -	65 A
9 	27 ESCAPE	46 .	66 B
10 LINEFEED	28 RED	47 /	67 C
11 DISABLE 	29 	48 0	68 D
12 ENABLE 	30 GREEN	49 1	69 E
13 	31 BLUE	50 2	70 F
14 LOWER CASE	32 SPACE	51 3	71 G
15 BLINK/FLASH ON	33 !	52 4	72 H
16 F9	34 "	53 5	73 I
17 	35 #	54 6	74 J
	36 \$	55 7	75 K
		56 8	76 L

77 M	106 □	135 F5	163 □
78 N	107 □	136 F7	164 □
79 O	108 □	137 F2	165 □
80 P	109 □	138 F4	166 ☒
81 Q	110 □	139 F6	167 □
82 R	111 □	140 F8	168 ☒
83 S	112 □	141 SHIFT RETURN	169 ☒
84 T	113 ☐	142 UPPERCASE	170 □
85 U	114 □	143 BLINK/FLASH OFF	171 ☒
86 V	115 ☐	144 BLACK	172 □
87 W	116 □	145 ↑	173 □
88 X	117 □	146 RVS OFF	174 □
89 Y	118 ☐	147 SHIFT CLR HOME	175 □
90 Z	119 ☐	148 SHIFT INST DEL	177 ☒
91 [120 ☐	149 BROWN	178 □
92 £	121 □	150 LT. RED	179 ☒
93]	122 ☐	151 DK. GRAY	180 □
94 ↑	123 ☐	152 GRAY	181 □
95 ←	124 ☐	153 LT. GREEN	182 □
96 ☐	125 ☐	154 LT. BLUE	183 □
97 ☒	126 π	155 LT. GRAY	184 □
98 ☠	127 ☐	156 PURPLE	185 □
99 ☐	128	157 ←	186 □
100 ☐	129 ORANGE	158 YELLOW	187 □
101 ☐	130 UNDERLINE OFF	159 CYAN	188 ☒
102 ☐	131	160 SPACE	189 ☒
103 ☠	132 HELP	161 ☐	190 □
104 ☠	133 F1	162 ☐	191 ☒
105 ☐	134 F3		

Note 1: Codes from 192 to 223 are equal to 96 to 127. Codes from 224 to 254 are equal to 160 to 190, and code 255 is equal to 126.

Note 2: While using lowercase/uppercase mode (by pressing +) , be aware that:

- The uppercase letters in region 65-90 of the above table are replaced with lowercase letters.
- The graphical characters in region 97-122 of the above table are replaced with uppercase letters.
- PETSCII's lowercase (65-90) and uppercase (97-122) letters are in ASCII's uppercase (65-90) and lowercase (97-122) letter regions.

CONTROL CODES

Keyboard Control	Function
Colours	
+ to	Choose from the first range of colours. More information on the colours available is under the BASIC BACKGROUND command on page B-25 .
+ to	Choose from the second range of colours.
+	Restores the colour of the cursor back to the default (white).
+	Switches the VIC-IV to colour range 0-15 (default colours). These colours can be accessed with and keys to (for the first 8 colours), or and keys to (for the remaining 8 colours).

Keyboard Control

Function

CTRL + A	Switches the VIC-IV to colour range 16-31 (alternate/rainbow colours). These colours can be accessed with CTRL and keys 1 to 8 (for the first 8 colours), or M and keys 1 to 8 (for the remaining 8 colours).
------------------------	---

Tabs

CTRL + Z	Tabs the cursor to the left. If there are no tab positions remaining, the cursor will remain at the start of the line.
CTRL + I	Tabs the cursor to the right. If there are no tab positions remaining, the cursor will remain at the end of the line.
CTRL + X	Sets or clears the current screen column as a tab position. Use CTRL + Z and I to jump back and forth to all positions set with X .

Movement

CTRL + Q	Moves the cursor down one line at a time. Equivalent to ↓ .
CTRL + J	Moves the cursor down a position. If you are on a long line of BASIC code that has extended to two lines, then the cursor will move down two rows to be on the next line.
CTRL + L	Equivalent to → .

Keyboard Control	Function
CTRL + T	Backspace the character immediately to the left and to shift all rightmost characters one position to the left. This is equivalent to INST DEL .
CTRL + M	Performs a carriage return, equivalent to RETURN .
Word movement	
CTRL + U	Moves the cursor back to the start of the previous word. If there are no words between the current cursor position and the start of the line, the cursor will move to the first column of the current line.
CTRL + W	Advances the cursor forward to the start of the next word. If there are no words between the cursor and the end of the line, the cursor will move to the first column of the next line.
Scrolling	
CTRL + P	Scroll BASIC listing down one line. Equivalent to F9 .
CTRL + V	Scroll BASIC listing up one line. Equivalent to F11 .
CTRL + S	Equivalent to NO SCROLL .
Formatting	
CTRL + B	Enables underline text mode. You can disable underline mode by pressing ESC , then O .

Keyboard Control	Function
CTRL + O	Enables flashing text mode. You can disable flashing mode by pressing ESC , then O .
Casing	
CTRL + N	Changes the text case mode from uppercase to lowercase.
CTRL + K	Locks the uppercase/lowercase mode switch usually performed with M + SHIFT .
CTRL + L	Enables the uppercase/lowercase mode switch that is performed with the M + SHIFT .
Miscellaneous	
CTRL + G	Produces a bell tone.
CTRL + [Equivalent to pressing ESC .
CTRL + *	Enters the Matrix Mode Debugger.

SHIFTED CODES

Keyboard Control	Function
SHIFT + INST DEL	Insert a character at the current cursor position and move all characters to the right by one position.
SHIFT + HOME	Clear home, clear the entire screen, and move the cursor to the home position.

ESCAPE SEQUENCES

To perform an Escape Sequence, press and release **esc**, then press one of the following keys to perform the sequence:

Key	Sequence
Editor behaviour	
esc	X
	Clears the screen and toggles between 40 and 80-column modes.
esc	4
	Clears the screen and switches to 40 column mode.
esc	8
	Clears the screen and switches to 80 column mode.
esc	@
	Clears a region of the screen, starting from the current cursor position, to the end of the screen.
esc	o
	Cancels the quote, reverse, underline, and flash modes.
Scrolling	
esc	V
	Scrolls the entire screen up one line.
esc	W
	Scrolls the entire screen down one line.
esc	L
	Enables scrolling when ↓ is pressed at the bottom of the screen.
esc	M
	Disables scrolling. When pressing ↓ at the bottom of the screen, the cursor will move to the top of the screen. However, when pressing ↑ at the top of the screen, the cursor will remain on the first line.
Insertion and deletion	

Key	Sequence
 	Inserts an empty line at the current cursor position and moves all subsequent lines down one position.
 	Deletes the current line and moves lines below the cursor up one position.
 	Erases all characters from the cursor to the start of the current line.
 	Erases all characters from the cursor to the end of the current line.
Movement	
 	Moves the cursor to the start of the current line.
 	Moves the cursor to the last non-whitespace character on the current line.
 	Saves the current cursor position. Use   (next to  Note that the  .
 	Restores the cursor position to the position stored via a prior a press of the   .
 	Restores the cursor position to the position stored via a prior a press of  .
Windowing	

Key	Sequence
 	<p>Sets the top-left corner of the windowed area. All typed characters and screen activity will be restricted to the area. Also see   . Windowed mode can be disabled by pressing  twice.</p>
 	<p>Sets the bottom right corner of the windowed area. All typed characters and screen activity will be restricted to the area. Also see   . Windowed mode can be disabled by pressing  twice.</p>
Cursor behaviour	
 	<p>Enables auto-insert mode. Any keys pressed will be inserted at the current cursor position, shifting all characters on the current line after the cursor to the right by one position.</p>
 	<p>Disables auto-insert mode, reverting back to overwrite mode.</p>
 	<p>Sets the cursor to non-flashing mode.</p>
 	<p>Sets the cursor to regular flashing mode.</p>
Bell behaviour	
 	<p>Enables the bell which can be sounded using  and .</p>
 	<p>Disable the bell so that pressing  and  will have no effect.</p>
Colours	

Key	Sequence		
 	Switches the VIC-IV to colour range 0-15 (default colours). These colours can be accessed with  and keys  to  and keys  to  	Switches the VIC-IV to colour range 16-31 (alternate/rainbow colours). These colours can be accessed with  and keys  to  and keys  to  Tabs <th data-kind="ghost"></th>	
 	Set the default tab stops (every 8 spaces) for the entire screen.		
 	Clears all tab stops. Any tabbing with  and  will move the cursor to the end of the line.		

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/Unicode (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  and pressing other keys or key-combinations. The use of ASCII or Unicode basic-latin 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 I/O 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 4 

Bit 1 Left 

Bit 5 

Bit 2

Bit 6 

Bit 3

Bit 7 Reserved

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.

Key Combination Tables

The following tables show the hex value and unicode character that will be produced by each key combination. One table is provided for pressing a key on its own, and one table for each of the , , , and  keys.

Keys pressed alone:

Key	Code	Unicode	Key	Code	Unicode	Key	Code	Unicode
INST DEL	\$14		RETURN	\$0D		→	\$1D	
F7	\$F7	÷	F1	\$F1	ñ	F3	\$F3	ó
F5	\$F5	õ	↓	\$11		3	\$33	3
W	\$77	w	A	\$61	a	4	\$34	4
Z	\$7A	z	S	\$73	s	E	\$65	e
SHIFT left	\$00		5	\$35	5	R	\$72	r
D	\$64	d	6	\$36	6	C	\$63	c
F	\$66	f	T	\$74	t	X	\$78	x
7	\$37	7	Y	\$79	y	G	\$67	g
8	\$38	8	B	\$62	b	H	\$68	h
U	\$75	u	V	\$76	v	9	\$39	9
I	\$69	i	J	\$6A	j	0	\$30	0
M	\$6D	m	K	\$6B	k	Ø	\$6F	ø
N	\$6E	n	+	\$2B	+	P	\$70	p
L	\$6C	l	-	\$2D	-	·	\$2E	.
:	\$3A	:	@	\$40	@	,	\$2C	,
£	\$A3	£	*	\$2A	*	;	\$3B	;
CLR HOME	\$13		SHIFT right	\$00		=	\$3D	=
↑	\$AF	-	/	\$2F	/	1	\$31	1
←	\$5F	-	CTRL	\$00		2	\$32	2
SPC	\$20		¶	\$00		Q	\$71	q
RUN STOP	\$03		NO SCROLL	\$00		TAB	\$09	
ALT	\$00		HELP	\$1F		F9	\$F9	ù
F11	\$FB	û	F13	\$FD	ý	ESC	\$1B	

Keys pressed with SHIFT

Key	Code	Unicode	Key	Code	Unicode	Key	Code	Unicode
INST DEL	\$94		RETURN	\$0D		→	\$9D	
F7	\$F8	ø	F1	\$F2	ö	F3	\$F4	ô
F5	\$F6	ö	↓	\$91		3	\$23	#
W	\$57	W	A	\$41	A	4	\$24	\$
Z	\$5A	Z	S	\$53	S	E	\$45	E
SHIFT left	\$00		5	\$25	%	R	\$52	R
D	\$44	D	6	\$26	&	C	\$43	C
F	\$46	F	T	\$54	T	X	\$58	X
7	\$27	'	Y	\$59	Y	G	\$47	G
8	\$28	(B	\$42	B	H	\$48	H
U	\$55	U	V	\$56	V	9	\$29)
I	\$49	I	J	\$4A	J	0	\$7B	{
M	\$4D	M	K	\$4B	K	O	\$4F	O
N	\$4E	N	+	\$00		P	\$50	P
L	\$4C	L	-	\$00		·	\$3E	>
:	\$5B	[@	\$00		,	\$3C	<
£	\$00		*	\$00		;	\$5D]
CLR HOME	\$93		SHIFT right	\$00		=	\$5F	-
↑	\$00		/	\$3F	?	1	\$21	!
←	\$60	`	CTRL	\$00		2	\$22	"
SPC	\$20		!@	\$00		Q	\$51	Q
RUN STOP	\$A3	£	NO SCROLL	\$00		TAB	\$0F	
ALT	\$00		HELP	\$1F		F9	\$FA	ú
F11	\$FC	ü	F13	\$FE	þ	ESC	\$1B	

Keys pressed with 

Key	Code	Unicode	Key	Code	Unicode	Key	Code	Unicode
INST DEL	\$94		RETURN	\$0D		→	\$9D	
F7	\$F8	ø	F1	\$F2	ö	F3	\$F4	ô
F5	\$F6	ö	↓	\$91		3	\$1C	
W	\$17		A	\$01		4	\$9F	
Z	\$1A		S	\$13		E	\$05	
SHIFT left	\$00		5	\$9C		R	\$12	
D	\$04		6	\$1E		C	\$03	
F	\$06		T	\$14		X	\$18	
7	\$1F		Y	\$19		G	\$07	
8	\$9E		B	\$02		H	\$08	
U	\$15		V	\$16		9	\$12	
I	\$09		J	\$0A		0	\$00	
M	\$0D		K	\$0B		O	\$0F	
N	\$0E		+	\$2B	+	P	\$10	
L	\$0C		-	\$2D	-	·	\$2E	.
:	\$3A	:	@	\$40	@	,	\$2C	,
£	\$00		*	\$EF	í	;	\$3B	;
CLR HOME	\$93		SHIFT right	\$00		=	\$3D	=
↑	\$00		/	\$2F	/	1	\$90	
←	\$60	~	CTRL	\$00		2	\$05	
SPC	\$20		!	\$00		Q	\$11	
RUN STOP	\$A3	£	NO SCROLL	\$00		TAB	\$0F	
ALT	\$00		HELP	\$1F		F9	\$FA	ú
F11	\$FC	ü	F13	\$FE	þ	ESC	\$1B	

Keys pressed with 

Key	Code	Unicode	Key	Code	Unicode	Key	Code	Unicode
INST DEL	\$94		RETURN	\$0D		→	\$ED	í
F7	\$F8	ø	F1	\$F2	ò	F3	\$F4	ô
F5	\$F6	ö	↓	\$EE	î	3	\$96	
W	\$D7	×	A	\$C1	Á	4	\$97	
Z	\$DA	Ú	S	\$D3	Ó	E	\$C5	À
SHIFT left	\$00		5	\$98		R	\$D2	Ò
D	\$C4	Ä	6	\$99		C	\$C3	Ã
F	\$C6	Æ	T	\$D4	Ô	X	\$D8	Ø
7	\$9A		Y	\$D9	Ù	G	\$C7	Ҫ
8	\$9B		B	\$C2	Â	H	\$C8	È
U	\$D5	Ö	V	\$D6	Ö	9	\$92	
I	\$C9	É	J	\$CA	Ê	0	\$81	
M	\$CD	Í	K	\$CB	Ë	O	\$CF	Ï
N	\$CE	Î	+	\$2B	+	P	\$D0	Đ
L	\$CC	Ї	-	\$2D	-	·	\$7C	
:	\$7B	{	@	\$40	@	,	\$7E	~
£	\$00		*	\$2A	*	;	\$7D	}
CLR HOME	\$93		SHIFT right	\$00		=	\$5F	-
↑	\$00		/	\$5C	\	1	\$81	
←	\$60	~	CTRL	\$00		2	\$95	
SPC	\$20			\$00		Q	\$D1	Ñ
RUN STOP	\$A3	¤	NO SCROLL	\$00		TAB	\$EF	í
ALT	\$00		HELP	\$1F		F9	\$FA	ú
F11	\$FC	ü	F13	\$FE	þ	ESC	\$1B	

Keys pressed with **ALT**

Key	Code	Unicode	Key	Code	Unicode	Key	Code	Unicode
INST DEL	\$7F		RETURN	\$00		→	\$DF	ß
F7	\$DE	þ	F1	\$B9	¹	F3	\$B2	²
F5	\$B3	³	↓	\$00		3	\$A4	¤
W	\$AE	®	A	\$E5	å	4	\$A2	¢
Z	\$F7	÷	S	\$A7	§	E	\$E6	æ
SHIFT left	\$00		5	\$B0	°	R	\$AE	®
D	\$F0	ð	6	\$A5	¥	C	\$E7	¤
F	\$00		T	\$FE	þ	X	\$D7	×
7	\$B4	'	Y	\$FF	ÿ	G	\$E8	è
8	\$E2	â	B	\$FA	ú	H	\$FD	ý
U	\$FC	ü	V	\$D3	Ó	9	\$DA	Ú
I	\$ED	í	J	\$E9	é	0	\$DB	Û
M	\$B5	þ	K	\$E1	á	O	\$F8	ø
N	\$F1	ñ	+	\$B1	±	P	\$B6	¶
L	\$F3	ó	-	\$AC	¬	·	\$BB	»
:	\$E4	ä	@	\$A8	..	,	\$AB	«
£	\$A3	£	*	\$B7	·	;	\$E4	ää
CLR HOME	\$DC	Ü	SHIFT right	\$DD	Ý	=	\$A6	¡
↑	\$AF	-	/	\$BF	¿	1	\$A1	i
←	\$B8	,	CTRL	\$00		2	\$AA	¤
SPC	\$AO		▼	\$00		Q	\$A9	©
RUN STOP	\$BA	¤	NO SCROLL	\$00		TAB	\$CO	À
ALT	\$00		HELP	\$1F		F9	\$BC	^{1/4}
F11	\$BD	%	F13	\$BE	[%]	ESC	\$DB	Û

Unicode Basic-Latin Keyboard Map

The following tables are a convenient reference to help you find a key combination that will produce the desired ASCII/Unicode basic-latin character code. Note that a very few codes are difficult to type in practice, because they are mapped to key combinations that perform other functions, in particular the combination of  +  is normally overridden by the Matrix Mode hardware debug feature.

		+\$00		+\$10		+\$20		+\$30
\$00	ALT	+	RETURN		F9		SPC	0 0
\$01			SHIFT right		↓	!	←	1 1
\$02			W		9	"	SPC	2 2
\$03			RUN STOP		CLR HOME	#	3	3 3
\$04			CTRL		INST DEL	\$	4	4 4
\$05			2		F9	%	R	5 5
\$06			F		F11	&	C	6 6
\$07			G		F11	'	Y	7 7
\$08			ALT		TAB	(B	8 8
\$09			TAB		F13)	9	9 9
\$0a			J		F13	*	*	:
\$0b			K		ESC	+	+	;
\$0c			L		£	,	,	< £
\$0d			RETURN		→	-	-	=
\$0e			N		↑	.	•	> :
\$0f			O	ALT	+	HELP	/	? 1

		+\$40		+\$50		+\$60		+\$70
\$00	C	©	P	P	`	←	p	P
\$01	A	A	Q	Q	a	A	q	Q
\$02	B	B	R	R	b	B	r	R
\$03	C	C	S	S	c	C	s	S
\$04	D	D	T	T	d	D	t	T
\$05	E	E	U	U	e	E	u	U
\$06	F	F	V	V	f	F	v	V
\$07	G	G	W	W	g	G	w	W
\$08	H	H	X	X	h	H	x	X
\$09	I	I	Y	Y	i	I	y	Y
\$0a	J	J	Z	Z	j	J	z	Z
\$0b	K	K	[©	k	K	{	:
\$0c	L	L	\	£	l	L		.
\$0d	M	M]	CLR HOME	m	M	}	;
\$0e	N	N	~	↑	n	N	~	,
\$0f	O	O	-	←	o	O		ALT + INST DEL

	+\$80	+\$90		+\$A0			+\$B0	
\$00		1		ALT	+ SPC	o	ALT	+ 5
\$01	1	↓	i	ALT	+ 1	±	ALT	+ +
\$02		0	¢	ALT	+ 4	2	ALT	+ F3
\$03	NO SCROLL	CLR HOME	£	ALT	+ £	3	ALT	+ F5
\$04	HELP	INST DEL	¤	ALT	+ 3	-	ALT	+ 7
\$05	F1	2	¥	ALT	+ 6	µ	ALT	+ M
\$06	F3	3	¡	ALT	+ =	¶	ALT	+ P
\$07	F5	4	§	ALT	+ S	.	ALT	+ *
\$08	F7	5	..	ALT	+ Ø	›	ALT	+ ←
\$09	F1	6	©	ALT	+ Q	1	ALT	+ F1
\$0a	F3	7	ª	ALT	+ 2	º	ALT	+ RUN STOP
\$0b	F5	8	«	ALT	+ ,	»	ALT	+ ·
\$0c	F7	5	¬	ALT	+ -	¼	ALT	+ F9
\$0d	RETURN	→			Z	½	ALT	+ F11
\$0e		8	®	ALT	+ W	¾	ALT	+ F13
\$0f		4	-	ALT	+ ↑	¿	ALT	+ /

		+\$CO		+\$DO		+\$EO		+\$FO	
\$00	À	[ALT]	+	TAB	Ð	P	à	õ	[ALT] + D
\$01	Á		A	Ñ	Q	á	[ALT] + K	ñ	[ALT] + N
\$02	Â		B	Ò	R	â	[ALT] + 8	ò	[M] + F1
\$03	Ã		C	Ó	[ALT] + V	ã		ó	[ALT] + L
\$04	Ä		D	Ö	T	ä	[ALT] + :	ô	[M] + F3
\$05	Å		E	Õ	U	å	[ALT] + A	õ	F5
\$06	Æ		F	Ö	V	æ	[ALT] + E	ö	[M] + F5
\$07	Ç		G	x	[ALT] + X	ç	[ALT] + C	÷	[ALT] + Z
\$08	È		H	Ø	X	è	[ALT] + G	ø	[ALT] + O
\$09	É		I	Ù	Y	é	[ALT] + J	ù	F9
\$0a	Ê		J	Ú	[ALT] + 9	ê		ú	[ALT] + B
\$0b	Ë		K	Û	[ALT] + 0	ë		û	F11
\$0c	Ì		L	Ü	[ALT] + CLR HOME	ì		ü	[ALT] + U
\$0d	Í		M	Ý	[ALT] + SHIFT right	í	[ALT] + I	ý	[ALT] + H
\$0e	Ï		N	Þ	[ALT] + F7	î	[M] + ↓	þ	[ALT] + T
\$0f	Ї		O	ß	[ALT] + →	ї	[M] + TAB	ÿ	[ALT] + Y

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 milli-second.

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:

	0	1	2	3	4	5	6	7	8
0	INST DEL	3	5	7	9	+	£	1	NO SCROLL
1	RETURN	W	R	Y	I	P	*	←	TAB
2	→	A	D	G	J	L	;	CTRL	ALT
3	F7	4	6	8	0	-	CLR HOME	2	HELP
4	F1	Z	C	B	M	.	SHIFT right	SPC	F9
5	F3	S	F	H	K	:	=	▼	F11
6	F5	E	T	U	O	@	↑	Q	F13
7	↓	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 in C64-mode. **CAPS
LOCK** 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, **CLR
HOME** 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 MEGApone 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. **RESTORE** exists outside of the keyboard matrix, as on the C64. To simulate holding **RESTORE** down, write \$52 (ASCII code for a capital R), and to simulate a quick tap of the **RESTORE**, write \$72 (ASCII code for a lowercase R). Another value must be written after the \$72 value has been written, if you wish to simulate multiple presses of **RESTORE**.

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 en-

sure \$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 program 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 FORI=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 FORI=0TO255:POKEDEC("D61E"),I:NEXT
100 FORI=255TO0STEP-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 FORI=0TO255:POKEDEC("D61E"),I:NEXT
150 FORI=255TO0STEP-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

4 <

1 9

5 INST
DEL

2 |

6 CLR
HOME

3 K

7	O	35	F
8	F3	36	V
9	8	37	SPACE
10	U	38	0
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	◎	55	↑ (next to RESTORE)
28	B	56	ESC
29	← (cursor left)	57	2
30	+	58	Q
31	:	59	A
32	NO SCROLL	60	Z
33	5	61	right SHIFT
34	R		

62  **F11**

63 *

64 Reserved

65 1

66 Reserved

67 Reserved

68 left  and 

69 /

70  **F9**

71 @

72  **RUN
STOP**

73  (next to 1)

74  **TAB**

75  **CTRL**

76  **M**

77 >

78  **F7**

79 P

APPENDIX

E

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 Chapter/Appendix [S on page S-3](#) as it contains a [Base Conversion](#) table for decimal, binary, and hexadecimal for integers between 0 and 255.

The BASIC listing for this appendix is a utility program that converts individual numbers into different bases. It can also convert multiple numbers within a specified range.

Although these concepts might be new now, with some practice they'll soon seem like second nature. We'll look at ways of expressing numbers in more detail. Later, we'll also investigate the various operations that we can perform on such numbers.

Decimal

When representing integers using decimal notation, each column in the number is for a different power of 10. The rightmost position represents the number of units (because $10^0 = 1$) and each column to the left of it is 10 times larger than the column before it. The rightmost column is called the units column. Columns to the left of it are labelled tens (because $10^1 = 10$), hundreds (because $10^2 = 100$), thousands (because $10^3 = 1000$), and so on.

To give an example, the integer 53280 represents the total of 5 lots of 10000, 3 lots of 1000, 2 lots of 100, 8 lots of 10 and 0 units. This can be seen more clearly if we break the integer up into distinct parts, by column.

Since

$$53280 = 50000 + 3000 + 200 + 80 + 0$$

we can present this as a table with the sum of each column at the bottom.

TEN THOUSANDS	THOUSANDS	HUNDREDS	TENS	UNITS
$10^4 = 10000$	$10^3 = 1000$	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
5	0	0	0	0
	3	0	0	0
		2	0	0
			8	0
				0
5	3	2	8	0

Another way of stating this is to write the expression using multiples of powers of 10.

$$53280 = (5 \times 10^4) + (3 \times 10^3) + (2 \times 10^2) + (8 \times 10^1) + (0 \times 10^0)$$

Alternatively

$$53280 = (5 \times 10000) + (3 \times 1000) + (2 \times 100) + (8 \times 10) + (0 \times 1)$$

We now introduce some useful terminology that is associated with decimal numbers.

The rightmost digit of a decimal number is called the least significant digit, because, being the smallest multiplier of a power of 10, it contributes the least to the number's magnitude. Each digit to the left of this digit has increasing significance. The leftmost (non-zero) digit of the decimal number is called the most significant digit, because, being the largest multiplier of a power of 10, it contributes the most to the number's magnitude.

For example, in the decimal number 53280, the digit 0 is the least significant digit and the digit 5 is the most significant digit.

A decimal number a is m orders of magnitude greater than the decimal number b if $a = b \times (10^m)$. For example, 50000 is three orders of magnitude greater than 50, because it has three more zeros. This terminology can be useful when making comparisons between numbers or when comparing the time efficiency or space efficiency of two programs with respect to the sizes of the given inputs.

Note that unlike binary (which uses a conventional % prefix) and hexadecimal (which uses a conventional \$ prefix), decimal numbers are given no special prefix. In some textbooks you might see such numbers with a subscript instead. So decimal numbers will have a 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

$$\begin{aligned} \%1101\ 0011 = \%1000\ 0000 + \%100\ 0000 + \%00\ 0000 + \%1\ 0000 + \%0000 + \%000 + \%10 \\ + \%1 \end{aligned}$$

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 nibble. See Chapter/Appendix [S on page S-3](#) 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 left-most 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.

$$\begin{array}{r} \% \ 1 \ 1 \ 0 \\ + \% \ 1 \ 1 \ 1 \\ \hline \% \ 1 \ 1 \ 0 \ 1 \end{array}$$

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.

$$\begin{array}{r} \$ 7 \ D \\ + \$ 6 \ 9 \\ \hline \$ E \ 6 \end{array}$$

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.

$$\begin{array}{r} \% 1 \ 0 \ 1 \ 1 \\ - \% \ 1 \ 1 \ 0 \\ \hline \% \ 1 \ 0 \ 1 \end{array}$$

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.

$$\begin{array}{r} \$ 3 \ D \\ - \$ 1 \ F \\ \hline \$ 1 \ E \end{array}$$

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 $\text{NOT } x$ 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 $x \text{ AND } y$ 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 $x \text{ OR } y$ 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 $x \text{ XOR } y$ 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.

	% 0 0 0 0	0 0 0 0	1 1 1 1	1 1 0 1
AND	% 0 0 0 0	0 0 0 0	1 1 1 0	1 1 1 0
	% 0 0 0 0	0 0 0 0	1 1 1 0	1 1 0 0

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 0 0	1 1 1 1 1	0 0 0 0 0
OR	% 0 0 0 0 0	0 0 0 0 0	0 0 0 1 1	0 0 1 1 1
	% 0 0 0 0 0	0 0 0 0 0	1 1 1 1 1	0 0 1 1 1

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	1	0	0	0	0	1	0	1	1
XOR	%	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	1
	%	0	0	0	1	0	1	1	0	1	1	0	0	0	1	1	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 switch off sprite 5 and switch 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 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 of 0 to 65535.

```

10 REM *****
20 REM *          *
30 REM *  INTEGER BASE CONVERTER  *
40 REM *          *
50 REM *****
60 POKE 0,65: BORDER 6: BACKGROUND 6: FOREGROUND 1
70 DIM P(15)
80 E$ = "STARTING INTEGER MUST BE LESS THAN OR EQUAL TO ENDING INTEGER"
90 FOR N = 8 TO 15
100 : P(N) = 2 ↑ N
110 NEXT N
120 REM *** OUTPUT MAIN MENU ***
130 PRINT CHR$(147)
140 PRINT: PRINT "INTEGER BASE CONVERTER"
150 L = 22: GOSUB 1930: PRINT L$
160 PRINT: PRINT "SELECT AN OPTION (S, M OR Q):: PRINT
170 PRINT "[S]{SPACE*2}SINGLE INTEGER CONVERSION"
180 PRINT "[M]{SPACE*2}MULTIPLE INTEGER CONVERSION"
190 PRINT "[Q]{SPACE*2}QUIT PROGRAM"
200 GET MS
210 IF (MS="S") THEN GOSUB 260: GOTO 140
220 IF (MS="M") THEN GOSUB 380: GOTO 140
230 IF (MS="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 B$
1590 :      IF B$="X" THEN RETURN
1600 :      IF B$="" THEN GOTO 1620
1610 :      GOTO 1580
1620 NEXT J
1630 PRINT CHR$(19): FOR I = 1 TO 22: PRINT: NEXT I
1640 PRINT SPC(20); "COMPLETE. PRESS SPACEBAR TO CONTINUE..."
1650 GET B$: IF B$<>" " THEN GOTO 1650: ELSE RETURN
1660 REM *** CONVERT MULTIPLE BINARY INTEGERS ***
1670 IB$=""
1680 PRINT: INPUT "ENTER STARTING BINARY INTEGER (UP TO 16 BITS): ", IB$
1690 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 1010: GOTO 1670
1780 DB = IB: DE = IE: DB$ = STR$(IB): DE$ = STR$(IE): GOTO 1380
1790 REM *** CONVERT MULTIPLE HEXADECIMAL INTEGERS ***
1800 HB$=""
1810 PRINT: INPUT "ENTER STARTING HEXADECIMAL INTEGER (UP TO 4 DIGITS): ", HB$
1820 HS=HB$: GOSUB 1220: HS=""": REM VALIDATE HEXADECIMAL INPUT
1830 IF (V = 0) THEN GOTO 1810
1840 HB = H
1850 HE$=""
1860 PRINT: INPUT "ENTER ENDING HEXADECIMAL INTEGER (UP TO 4 DIGITS): ", HE$
1870 HS=HE$: GOSUB 1220: HS=""": REM VALIDATE HEXADECIMAL INPUT
1880 IF (V = 0) THEN GOTO 1860
1890 HE = H
1900 IF (HE < HB) THEN AS = E$: GOSUB 1010: GOTO 1880
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

```


APPENDIX



System Memory Map

- **Introduction**
- **MEGA65 Native Memory Map**
- **\$D000 - \$DFFF I/O 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 256MB 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 flexible memory map.

In this appendix, we will begin by describing the MEGA65's native memory map, that is, where all of the memory, I/O 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 MB of memory, i.e., 16 memory banks of 64KB each. This is because the C65's 4510 CPU can access only 1MB of address space. These banks can be accessed from BASIC 65 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 64KB RAM. This is the RAM visible in C64-mode.
1	1	\$1xxxx	Second 64KB RAM. This is the 2nd 64KB 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 (384KB or larger chip-RAM models)
5	5	\$5xxxx	Additional RAM (384KB or larger chip-RAM models)
6	6	\$6xxxx	Additional RAM (*512KB or larger chip-RAM models)
7	7	\$7xxxx	Additional RAM (*512KB or larger chip-RAM models)
8	8	\$8xxxx	Additional RAM (*1MB or larger chip-RAM models)
9	9	\$9xxxx	Additional RAM (*1MB or larger chip-RAM models)
A	10	\$Axxxx	Additional RAM (*1MB or larger chip-RAM models)
B	11	\$Bxxxx	Additional RAM (*1MB or larger chip-RAM models)
C	12	\$Cxxxx	Additional RAM (*1MB or larger chip-RAM models)
D	13	\$Dxxxx	Additional RAM (*1MB or larger chip-RAM models)

continued ...

...continued

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

* Note that the MEGA65 presently only provides a model featuring 384KB of chip-RAM. Future models may feature larger amounts of chip-RAM (such as 512KB and 1MB).

The key features of this address space are the 128KB of RAM in the first two banks, which is also present on the C65. If you intend to write programs 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 128KB 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 program 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 programs. This gives a total of 192KB of RAM, which is available on all models of the MEGA65.

On models that have 384KB or more of chip RAM, BANK 4 and 5 are also available. Similarly, models which provide 1MB or more of chip RAM will have BANK 6 through 15 also available, giving a total of 896KB (or 960KB, if only the C64 part of the ROM is required) of RAM available for your programs. Note that the MEGA65's built-in freeze cartridge currently freezes only the first 384KB 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 32KB or 64KB 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.

Additional RAM

Apart from the 384kb of chip-RAM found as standard on all MEGA65 models, most models (devkit, release boards and xemu, but NOT on Nexys boards currently) also have an extra 8MB of RAM starting at \$8000000, referred to as 'ATTIC RAM'. It is not visible to the other chips (vic/sid/etc) and can't be used for audio DMA, but code can run from it (more slowly) or it can be used to store content and DMA it in/out of the chip-RAM.

There are also plans underway to support a PMOD hyperRAM module (installed via the trapdoor beneath the MEGA65) in order to provide a further 8MB of RAM starting at \$8800000, referred to as 'CELLAR RAM'.

28-bit Address Space

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

HEX	DEC	Size	Contents
0000000	0	1	CPU I/O Port Data Direction Register
0000001	1	1	CPU I/O Port Data
0000002 - 005FFFF	2 - 384KB	384KB	Fast chip RAM (40MHz)
0060000 - 0FFFFFF	384KB - 16MB	15.6MB	Reserved for future chip RAM expansion
1000000 - 3FFFFFF	16MB - 64MB	48MB	Reserved
4000000 - 7FFFFFF	64MB - 128MB	64MB	Cartridge port and other devices on the slow bus (1 - 10 MHz)
8000000 - 87FFFFFF	128MB - 135MB	8MB	8MB ATTIC RAM (all models apart from Nexys, presently)
8800000 - 8FFFFFF	135MB - 144MB	8MB	8MB CELLAR RAM (planned PMOD module installed via trapdoor)
9000000 - EFFFFFF	144MB - 240MB	96MB	Reserved for future expansion RAM
F000000 - FF7DFFF	240MB - 255.49MB	15.49MB	Reserved for future I/O expansion

continued ...

...continued

HEX	DEC	Size	Contents
FF7E000 - FF7FFFF	255.49MB - 255.49MB	4KB	VIC-IV Character ROM (write only)
FF80000 - FF87FFF	255.5MB - 255.53MB	32KB	VIC-IV Colour RAM (32KB colour RAM - available on all models)
FF88000 - FF8FFFF	255.53MB - 255.57MB	32KB	Additional VIC-IV Colour RAM (64KB colour RAM - planned to be available on R3 models and beyond)
FF90000 - FFCAFFF	255.53MB - 255.80MB	216KB	Reserved
FFCB000 - FFCBFFF	255.80MB - 255.80MB	4KB	Emulated C1541 RAM
FFCC000 - FFCFFFF	255.80MB - 255.81MB	16KB	Emulated C1541 ROM
FFD0000 - FFD0FFF	255.81MB - 255.81MB	4KB	C64 \$Dxxx I/O Personality
FFD1000 - FFD1FFF	255.81MB - 255.82MB	4KB	C65 \$Dxxx I/O Personality
FFD2000 - FFD2FFF	255.82MB - 255.82MB	4KB	MEGA65 \$Dxxx Ethernet I/O Personality
FFD3000 - FFD3FFF	255.82MB - 255.82MB	4KB	MEGA65 \$Dxxx Normal I/O Personality
FFD4000 - FFD5FFF	255.82MB - 255.83MB	8KB	Reserved
FFD6000 - FFD67FF	255.83MB - 255.83MB	2KB	Hypervisor scratch space
FFD6000 - FFD6BFF	255.83MB - 255.83MB	3KB	Hypervisor scratch space
FFD6C00 - FFD6dff	255.83MB - 255.83MB	512	F011 floppy controller sector buffer
FFD6E00 - FFD6FFF	255.83MB - 255.83MB	512	SD Card controller sector buffer
FFD7000 - FFD70FF	255.83MB - 255.83MB	256	MEGApone r1 I2C peripherals
FFD7100 - FFD71FF	255.83MB - 255.83MB	256	MEGA65 r2 I2C peripherals
FFD7200 - FFD72FF	255.83MB - 255.83MB	256	MEGA65 HDMI I2C registers (only for R2 and older models fitted with the ADV7511 HDMI driver chip)

continued ...

...continued

HEX	DEC	Size	Contents
FFD7300 - FFD7FFF	255.83MB - 255.84MB	3.25KB	Reserved for future I2C peripherals
FFD8000 - FFDBFFF	255.83MB - 255.86MB	16KB	Hypervisor ROM (only visible in Hypervisor Mode)
FFDC000 - FFDDFFF	255.86MB - 255.87MB	8KB	Reserved for Hypervisor Mode ROM expansion
FFDE000 - FFDE7FF	255.87MB - 255.87MB	2KB	Reserved for Ethernet buffer expansion
FFDE800 - FFDEFFF	255.87MB - 255.87MB	2KB	Ethernet frame read buffer (read only) and Ethernet frame write buffer (write only)
FFDF000 - FFDFFFF	255.87MB - 255.87MB	4KB	Virtual FPGA registers (selected models only)
FFE0000 - FFFFFFFF	255.87MB - 256MB	128KB	Reserved

\$D000 - \$DFFF I/O PERSONALITIES

The MEGA65 supports four different I/O personalities. These are selected by writing the appropriate values to the \$D02F KEY register, which is visible in all four I/O personalities. There is more information in Chapter/Appendix [10 on page 10-3](#) about the use of the KEY register.

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

HEX	C64	C65	MEGA65 ETHERNET	MEGA65
KEY	\$00	\$A5, \$96	\$45, \$54	\$47, \$53
\$D000 - \$D02F	VIC-II	VIC-II	VIC-II	VIC-II
\$D030 - \$D07F	VIC-II ¹	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 I/O	CART I/O	ETHERNET Buffer	CART I/O / SD SECTOR

¹ In the C64 I/O personality, \$D030 behaves as on C128, allowing toggling between 1MHz and 2MHz CPU speed.

² The additional MEGA65 SIDs are visible in all I/O personalities.

³ 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 I/O 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.

I/O/CART	CART ROMLO	CART ROMHI		I/O	CART ROMHI	
C65	BASIC	BASIC	INTER-FACE		KERNEL	
MAP	MAP LO (4 x 8KB slabs)		MAP HI (4 x 8KB slabs)			
C64		BASIC		CHAR ROM	KERNEL	
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™ 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 KERNEL and BASIC system. The MEGA65 is different: It uses 128KB of its 384KB fast chip RAM at \$20000 - \$3FFF (banks 2 and 3) to hold these system programs. 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 program 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 program 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 I/O 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.

NOTE: If you are starting your program from C65-mode, you must first make sure that the I/O area is visible at \$D000-\$DFFF. The simplest way to do this is to use the MAP instruction with all zero values in the registers. The following fragment demonstrates

this, and also makes sure that the MEGA65 I/O context is active, so that the hypervisor trap will be able to trigger:

```
; Clear C65 memory map
LDA #$00
TAX
TAY
TAZ
MAP
; Bank I/O in via C64 mechanism
LDA #$35
STA $01
; Do MEGA65 / VIC-IV I/O knock
LDA #$47
STA $D02F
LDA #$53
STA $D02F
; End MAP sequence, thus allowing interrupts to occur again
EOM
; Do Hypervisor call to un-write-protect the ROM area
LDA #$70
STA $D640
NOP
```

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 KERNAL
\$3C000 -- \$3DFFF	RESERVED
\$38000 -- \$3BFFF	C65 BASIC GRAPHICS ROUTINES
\$32000 -- \$37FFF	C65 BASIC
\$30000 -- \$31FFF	MONITOR (gets mapped at \$6000 -- \$7FFF)
\$2E000 -- \$2FFFF	C64 KERNAL
\$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 Acceleration 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 I/O registers of the 45GS02.

DIFFERENCES TO THE 6502

The 45GS02 has a number of key differences to earlier 6502-class processors:

Supervisor/Hypervisor Mode **Privileged**

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 16KB 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 I/O-mode at \$D640 - \$D67F are replaced by the virtualisation control registers. These registers 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 Unintended Instructions

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 or unintended 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 unintended 6502 instructions 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.

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 Chapter/Appendix [L](#) on page [L-5](#)

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 - \$1FFFF, \$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
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 Chapter/Appendix [M on page M-5](#).

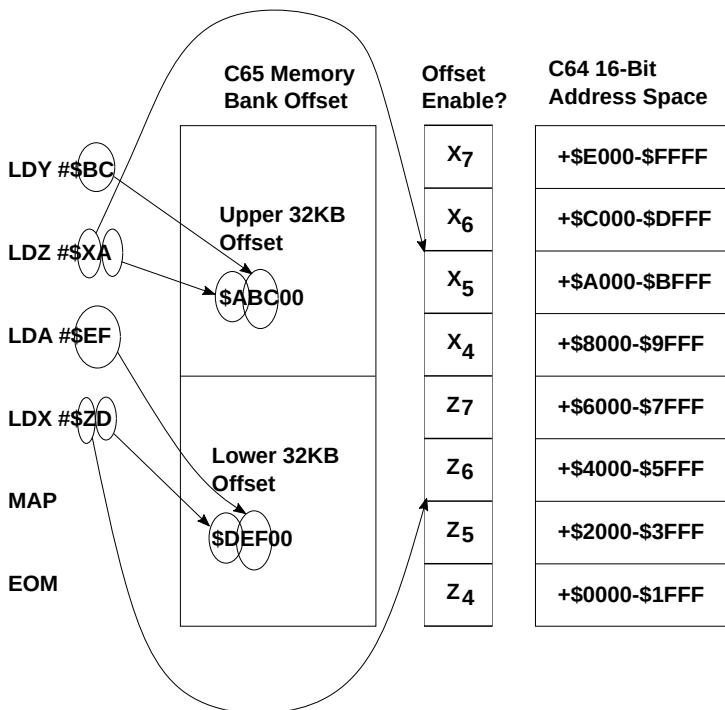
The MAP instruction

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 processors 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.

The following diagram illustrates how the MAP instruction takes the values of the four A, X, Y and Z registers, and uses them to compute the upper and lower address offsets, and sets the bank enable bits for each of the eight 8KB memory regions of the 6502 address space:

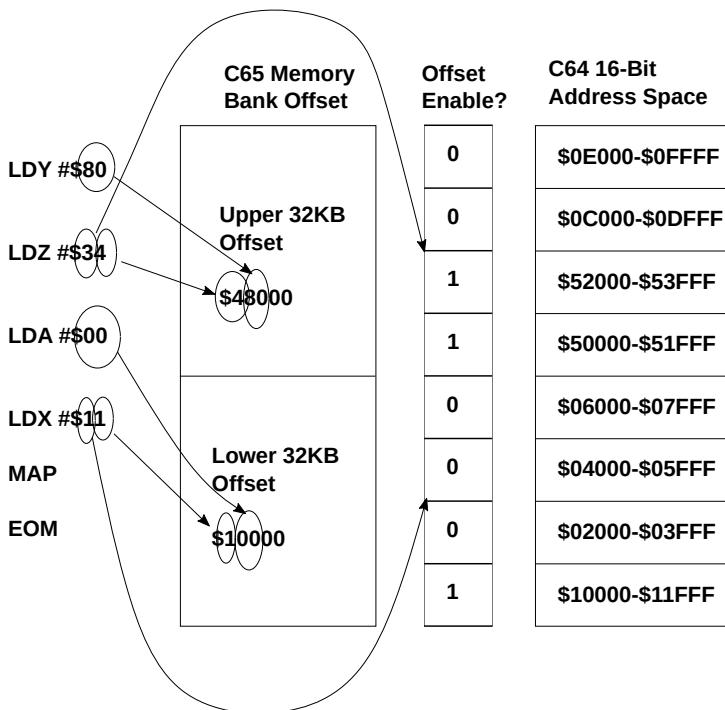


That is, the contents of the A register and the lower-nibble of the X register form a 12-bit value that is multiplied by 256 to produce the offset used for any of the 8KB banks in the lower 32KB half of the 6502's 16-bit address space. The upper nibble of the X register is used as flags to indicate which of the four 8KB blocks in that 32KB half of the 6502 address space should have the offset added to their addresses to compute the actual address.

The Y and Z registers are used in a similar way to produce the offset for the upper 32KB half of the 6502 address space, and the flags to indicate whether the offset is used for each of the four 8KB blocks in that half of the address space.

Note that the lower 8 bits of the offset cannot be set. That is, the offset will be a multiple of 256 bytes, unlike on some extended 6502 processors. However, in practice this restriction is rarely limiting.

To understand how this works in practice, the following example shows how this works with a concrete example, showing the address ranges that would be visible in each of the 8KB slices of the 6502's 64KB address space:



Notice that the offsets for each of the two 32KB address ranges get added to the 6502 address. This is why the offset of \$48000 for the upper 32KB generates an address of \$50000 at the 6502 address \$8000.

See also under "Using the MAP instruction to access >1MB" for further explanation.

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 Chapter/Appendix [L on page L-5](#)

Flat Memory Access

Accessing memory beyond the 1MB point

The MEGA65 can support up to 256MB of memory. This is more than the 1MB 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.

As another 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 in the bottom 64KB of RAM, and not in the region between \$6000 - \$8000.

As the MAP instruction normally is only aware of the C65-style 20-bit addresses, the MEGA65 extension to the instruction must be used to set the upper 8 bits of the 28-bit MEGA65 addresses, i.e., which mega-byte of address space should be used for the address translation. This is done by setting the X register to \$0F when setting the mega-byte number for the lower-32KB of the C64-style 64KB address space. This does not create any incompatibility with any sensible use of the MAP instruction on a C65, because this value indicates that none of the four 8KB memory blocks will be remapped, 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 \$FFxxxxx 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 - $FFDEFFFF, so select $FF megabyte section  
LDA #$ff  
LDX #$0f  
LDY #$00  
LDZ #$00  
Map  
  
; now enable mapping of $DE000-$DFFFF at $6000  
; M6Ps are offset based, so we need to subtract $6000 from the target address  
; $DE000 - $6000 = $D8000  
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
```

If you are using the ACME assembler, or another assembler that supports the 45GS02 extensions, you can instead use square-brackets to indicate that you are performing a flat-memory operation. Such assemblers will insert the \$EA prefix automatically for you. For example:

```
LDA [$45],Z
```

Regardless which tool you are using, this example 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 34 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 (A is LSB, Z is MSB) as a single virtual 32-bit register, also called the *Q pseudo 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  
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 required  
STA $1240  
STX $1241
```

This approach works with the LDA, STA, ADC, SBC, CMP, EOR, AND, BIT, ORA, ASL, ASR, LSR, ROL, ROR, INC and DEC instructions. If you are using ACME or another 45GS02 aware assembler, you can instead use the new **LDQ**, **STQ**, **ADQC**, **SBCQ**, **CPQ**, **EORQ**, **ANDQ**, **BITQ**, **ORQ**, **ASLQ**, **ASRQ**, **LSRQ**, **ROLQ**, **RORQ**, **INQ** and **DEQ** mnemonics. The previous example would thus become:

```

; Clear carry before performing addition, as normal
CLC
LDQ $1234 ; Load the contents of $1234-$1237 into A,X,Y and Z respectively
ADCQ $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

STQ $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 r
STA $1240
STX $1241

```

The virtual 32-bit addressing mode works with any addressing mode. However, 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 exception is the Zero-Page Indirect Z-Indexed addressing mode: In this case the Z register is NOT added to the target address (with the exception of the LDQ opcode), unlike 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, e.g., to load a constant into the four general purpose registers, or to add or subtract a constant value. This is to avoid problems with variable length instructions.

For LDQ and STQ, it would save at most one byte compared to LDA #\$nn ... LDZ #\$nn, and would be no faster. In fact, for many common values, such as #\$00000000, there are short-cuts, such as:

```

LDA #$00
TAX
TAY
TAZ

```

If you need to add or subtract a 32-bit immediate value, this may require you to re-order the arguments, or perform other minor gymnastics. For example, to compute the sum of the contents of memory and an immediate value, you can load the A, X, Y and Z registers with the immediate value, and then use **ADCQ** with the memory address, e.g.:

```

; Get the immediate value #$12345678 into Q
LDA #$78
LDX #$56
LDY #$34
LDZ #$12
; Add the contents of memory locations $1234-$1237
NEG
NEG
ADC $1234
; Store the result back in $1234-$1237
NEG
NEG
STA $1234

```

Again, if you are using the ACME or another 45GS02-aware assembler, this can be more compactly and clearly written as follows. But note that in both cases the same byte-sequence of machine code is produced, and the program will take the same number of cycles to execute.

```

; Get the immediate value #$12345678 into Q
LDA #$78
LDX #$56
LDY #$34
LDZ #$12
; Add the contents of memory locations $1234-$1237
ADQC $1234
; Store the result back in $1234-$1237
STQ $1234

```

C64 CPU MEMORY MAPPED REGISTERS

HEX	DEC	Signal	Description
00	0	PORDDR	6510/45GS10 CPU port DDR
01	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				
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				

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
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			
D67E	54910					HTRAP3E			
D67F	54911					HTRAP3F			
D710	55056	-		BADEXTRA	BRCOST	-	SLIEN	BADLEN	
D7EF	55279			RAND					
D7FA	55290				FRAMECOUNT				
D7FB	55291			-		CARTEN	-		
D7FD	55293	NOEXROM	NOGAME		-			POWEREN	
D7FE	55294			-		OCEANA	PREFETCH		

- **BADEXTRA** Cost of badlines minus 40. ie. 00=40 cycles, 11 = 43 cycles.
- **BADLEN** Enable badline emulation
- **BRCOST** 1=charge extra cycle(s) for branches taken
- **CARTEN** 1= enable cartridges
- **FRAMECOUNT** Count number of elapsed video frames
- **HTRAPXX** Writing triggers hypervisor trap \$XX
- **NOEXROM** Override for /EXROM : Must be 0 to enable /EXROM signal
- **NOGAME** Override for /GAME : Must be 0 to enable /GAME signal
- **OCEANA** Enable Ocean Type A cartridge emulation
- **POWEREN** Set to zero to power off computer on supported systems. WRITE ONLY.
- **PREFETCH** Enable expansion RAM pre-fetch logic
- **RAND** Hardware random number generator
- **SLIEN** Enable 6502-style slow (7 cycle) interrupts

MEGA65 CPU MATHS ACCELERATION REGISTERS

Every MEGA65 contains a combined 32-bit hardware multiplier and divider. This device takes two 32-bit inputs, **MULTINA** and **MULTINB**, and simultaneously calculates:

- the 64-bit product **MULTOUT** of **MULTINA** and **MULTINB**
- the 32-bit whole part **DIVOUT(4-7)** of **MULTINA** divided by **MULTINB**
- the 32-bit fractional part **DIVOUT(0-3)** of **MULTINA** divided by **MULTINB**

It is always updating the outputs based on the inputs, so there is no need to take special action when changing the inputs. The multiplier takes 1 cycle to calculate, and the updated result will thus be available immediately (a **MULBUSY** bit is defined, but currently it won't be set at all). The hardware divider, however, can take upto 20 cycles depending on the particular inputs. The programmer should check the **DIVBUSY** bit if the divider is still calculating:

```
loop: BIT $D70F ; transfer DIVBUSY bit into N flag
      BMI loop ; as long as it is set, we need to wait
```

The MEGA65 is planned to also include a programmable math unit, which helps to accelerate the calculation of fixed-point formulae. This is presently disabled and will be further documented if and when it becomes available (addresses \$D780 - \$D7E3).

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D70F	55055	DIVBUSY	MULBUSY				-		
D768	55144				DIVOUT				
D769	55145				DIVOUT				
D76A	55146				DIVOUT				
D76B	55147				DIVOUT				
D76C	55148				DIVOUT				
D76D	55149				DIVOUT				
D76E	55150				DIVOUT				
D76F	55151				DIVOUT				
D770	55152				MULTINA				
D771	55153				MULTINA				
D772	55154				MULTINA				
D773	55155				MULTINA				
D774	55156				MULTINB				
D775	55157				MULTINB				
D776	55158				MULTINB				
D777	55159				MULTINB				
D778	55160				MULTOUT				
D779	55161				MULTOUT				
D77A	55162				MULTOUT				
D77B	55163				MULTOUT				
D77C	55164				MULTOUT				
D77D	55165				MULTOUT				
D77E	55166				MULTOUT				
D77F	55167				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				

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D78A	55178					MATHIN2			
D78B	55179					MATHIN2			
D78C	55180					MATHIN3			
D78D	55181					MATHIN3			
D78E	55182					MATHIN3			
D78F	55183					MATHIN3			
D790	55184					MATHIN4			
D791	55185					MATHIN4			
D792	55186					MATHIN4			
D793	55187					MATHIN4			
D794	55188					MATHIN5			
D795	55189					MATHIN5			
D796	55190					MATHIN5			
D797	55191					MATHIN5			
D798	55192					MATHIN6			
D799	55193					MATHIN6			
D79A	55194					MATHIN6			
D79B	55195					MATHIN6			
D79C	55196					MATHIN7			
D79D	55197					MATHIN7			
D79E	55198					MATHIN7			
D79F	55199					MATHIN7			
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					MATHINA			
D7A9	55209					MATHINA			
D7AA	55210					MATHINA			
D7AB	55211					MATHINA			
D7AC	55212					MATHINB			
D7AD	55213					MATHINB			
D7AE	55214					MATHINB			
D7AF	55215					MATHINB			

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D7B0	55216					MATHINC			
D7B1	55217					MATHINC			
D7B2	55218					MATHINC			
D7B3	55219					MATHINC			
D7B4	55220					MATHIND			
D7B5	55221					MATHIND			
D7B6	55222					MATHIND			
D7B7	55223					MATHIND			
D7B8	55224					MATHINE			
D7B9	55225					MATHINE			
D7BA	55226					MATHINE			
D7BB	55227					MATHINE			
D7BC	55228					MATHINF			
D7BD	55229					MATHINF			
D7BE	55230					MATHINF			
D7BF	55231					MATHINF			
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			UNITAINB				UNITAINA	
D7CB	55243			UNITBINB				UNITBINA	
D7CC	55244			UNITCINB				UNITCINA	
D7CD	55245			UNITDINB				UNITDINA	
D7CE	55246			UNITEINB				UNITEINA	
D7CF	55247			UNITFINB				UNITFINA	
D7D0	55248	U0LATCH	U0MLADD	U0HIOUT	U0-LOWOUT			UNIT0OUT	
D7D1	55249	U1LATCH	U1MLADD	U1HIOUT	U1-LOWOUT			UNIT1OUT	
D7D2	55250	U2LATCH	U2MLADD	U2HIOUT	U2-LOWOUT			UNIT2OUT	

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D7D3	55251	U3LATCH	U3MLADD	U3HIOUT	U3-LOWOUT		UNIT3OUT		
D7D4	55252	U4LATCH	U4MLADD	U4HIOUT	U4-LOWOUT		UNIT4OUT		
D7D5	55253	U5LATCH	U5MLADD	U5HIOUT	U5-LOWOUT		UNIT5OUT		
D7D6	55254	U6LATCH	U6MLADD	U6HIOUT	U6-LOWOUT		UNIT6OUT		
D7D7	55255	U7LATCH	U7MLADD	U7HIOUT	U7-LOWOUT		UNIT7OUT		
D7D8	55256	U8LATCH	U8BSADD	U8HIOUT	U8-LOWOUT		UNIT8OUT		
D7D9	55257	U9LATCH	U9BSADD	U9HIOUT	U9-LOWOUT		UNIT9OUT		
D7DA	55258	UALATCH	UABSADD	UAHIOUT	UA-LOWOUT		UNITAOUT		
D7DB	55259	UBLATCH	UBBSADD	UBHIOUT	UB-LOWOUT		UNITBOUT		
D7DC	55260	UCLATCH	UCDVADD	UCHIOUT	UC-LOWOUT		UNITCOUT		
D7DD	55261	UDLATCH	UDDVADD	UDHIOUT	UD-LOWOUT		UNITDOUT		
D7DE	55262	UELATCH	UEDVADD	UEHIOUT	UE-LOWOUT		UNITEOUT		
D7DF	55263	UFLATCH	UFDVADD	UFHIOUT	UF-LOWOUT		UNITFOUT		
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)
- **DIVBUSY** Set if hardware divider is busy
- **DIVOUT** 64-bit output of $MULTINA \div MULTINB$
- **LATCHINT** Latch interval for latched outputs (in CPU cycles)
- **MATHINX** Math unit 32-bit input X
- **MULBUSY** Set if hardware multiplier is busy
- **MULTINA** Multiplier input A / Divider numerator (32 bit)

- **MULTINB** Multiplier input B / Divider denominator (32 bit)
- **MULTOUT** 64-bit output of MULTINA × MULTINB
- **RESERVED** Reserved
- **UNITXINA** Select which of the 16 32-bit math registers is input A for Math Function Unit X.
- **UNITXINB** Select which of the 16 32-bit math registers is input B for Math Function Unit X.
- **UNITXOUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit X
- **UXBSADD** If set, Math Function Unit Y acts as a 32-bit adder instead of 32-bit barrel-shifter.
- **UXDVADD** If set, Math Function Unit X acts as a 32-bit adder instead of 32-bit divider.
- **UXHIOUT** If set, the high-half of the output of Math Function Unit X is written to math register UNITXOUT.
- **UXLATCH** If set, Math Function Unit X's output is latched.
- **UXLOWOUT** If set, the low-half of the output of Math Function Unit X is written to math register UNITXOUT.
- **UXMLADD** If set, Math Function Unit X acts as a 32-bit adder instead of 32-bit multiplier.
- **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 16KB 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. It's 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. It can, however, always be accessed from the serial monitor/debugger interface via its 28-bit address, \$FFF8000 - \$FFF8FFFF. 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 I/O 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  for >0.5 seconds, or the  and  key combination traps to the hypervisor. Typically the first is used to launch the Freeze Menu and the second to toggle the display of the debug interface.
- **Accessing virtualised I/O devices** For example, if the F011 (internal 3.5" disk drive controller) has been virtualised, then attempting to read or write sectors using this device will cause traps to the hypervisor.
- **Executing an instruction that would lock up the CPU** A number of undocumented opcodes on the 6502 will cause the CPU to lockup. On the MEGA65, instead of locking up, the computer will trap to the hypervisor. This could be used to implement alternative instruction behaviours, or simply to tell the user that something bad has happened.
- **Certain special events** Some devices can generate hypervisor-level interrupts. These are implemented as traps to the hypervisor.

The 45GS02 handles all of these in a similar manner internally:

1. The SYSCALL or trap address is calculated, based on the event.
2. The contents of all CPU registers are saved into the virtualisation control registers.

3. The hypervisor mode memory layout is activated, the CPU decimal flag and special purpose registers are all set to appropriate values. The contents of the A,X,Y and Z and most other CPU flags are preserved, so that they can be accessed from the Hypervisor's SYSCALL/trap handler routine, without having to load them, thus saving a few cycles for each call.
4. The hypervisor-mode flag is asserted, and the program 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 [2].

Hypervisor Memory Layout

The hypervisor memory is 16KB in size. The first 512 bytes are reserved for SYSCALL and system trap entry points, with four bytes for each. For example, the reset entry point is at \$8100 - \$8100 + 3 = \$8100 - \$8103. This allows 4 bytes for an instruction, typically a JMP instruction, followed by a NOP to pad it to 4 bytes.

The full list of SYSCALLs and traps is:

HEX	DEC	Name	Description
8000	32768	SYSCALL00	SYSCALL 0 entry point
8004	32772	SYSCALL01	SYSCALL 1 entry point
8008	32776	SYSCALL02	SYSCALL 2 entry point
800C	32780	SYSCALL03	SYSCALL 3 entry point
8010	32784	SYSCALL04	SYSCALL 4 entry point
8014	32788	SYSCALL05	SYSCALL 5 entry point
8018	32792	SYSCALL06	SYSCALL 6 entry point
801C	32796	SYSCALL07	SYSCALL 7 entry point
8020	32800	SYSCALL08	SYSCALL 8 entry point
8024	32804	SYSCALL09	SYSCALL 9 entry point
8028	32808	SYSCALL0A	SYSCALL 10 entry point
802C	32812	SYSCALL0B	SYSCALL 11 entry point
8030	32816	SYSCALL0C	SYSCALL 12 entry point
8034	32820	SYSCALL0D	SYSCALL 13 entry point
8038	32824	SYSCALL0E	SYSCALL 14 entry point
803C	32828	SYSCALL0F	SYSCALL 15 entry point

continued ...

...continued

HEX	DEC	Name	Description
8040	32832	SYSCALL10	SYSCALL 16 entry point
8044	32836	SECURENTR	Enter secure container trap entry point
8048	32840	SECUREXIT	Leave secure container trap entry point.
804C	32844	SYSCALL13	SYSCALL 19 entry point
8050	32848	SYSCALL14	SYSCALL 20 entry point
8054	32852	SYSCALL15	SYSCALL 21 entry point
8058	32856	SYSCALL16	SYSCALL 22 entry point
805C	32860	SYSCALL17	SYSCALL 23 entry point
8060	32864	SYSCALL18	SYSCALL 24 entry point
8064	32868	SYSCALL19	SYSCALL 25 entry point
8068	32872	SYSCALL1A	SYSCALL 26 entry point
806C	32876	SYSCALL1B	SYSCALL 27 entry point
8070	32880	SYSCALL1C	SYSCALL 28 entry point
8074	32884	SYSCALL1D	SYSCALL 29 entry point
8078	32888	SYSCALL1E	SYSCALL 30 entry point
807C	32892	SYSCALL1F	SYSCALL 31 entry point
8080	32896	SYSCALL20	SYSCALL 32 entry point
8084	32900	SYSCALL21	SYSCALL 33 entry point
8088	32904	SYSCALL22	SYSCALL 34 entry point
808C	32908	SYSCALL23	SYSCALL 35 entry point
8090	32912	SYSCALL24	SYSCALL 36 entry point
8094	32916	SYSCALL25	SYSCALL 37 entry point
8098	32920	SYSCALL26	SYSCALL 38 entry point
809C	32924	SYSCALL27	SYSCALL 39 entry point
80A0	32928	SYSCALL28	SYSCALL 40 entry point
80A4	32932	SYSCALL29	SYSCALL 41 entry point
80A8	32936	SYSCALL2A	SYSCALL 42 entry point
80AC	32940	SYSCALL2B	SYSCALL 43 entry point
80B0	32944	SYSCALL2C	SYSCALL 44 entry point
80B4	32948	SYSCALL2D	SYSCALL 45 entry point
80B8	32952	SYSCALL2E	SYSCALL 46 entry point
80BC	32956	SYSCALL2F	SYSCALL 47 entry point
80C0	32960	SYSCALL30	SYSCALL 48 entry point
80C4	32964	SYSCALL31	SYSCALL 49 entry point
80C8	32968	SYSCALL32	SYSCALL 50 entry point
80CC	32972	SYSCALL33	SYSCALL 51 entry point
80D0	32976	SYSCALL34	SYSCALL 52 entry point
80D4	32980	SYSCALL35	SYSCALL 53 entry point

continued ...

...continued

HEX	DEC	Name	Description
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 16KB 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:

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

continued ...

...continued

HEX	DEC	Description
8E00 - BEFF	48640 - 48895	Processor stack for hypervisor or operating system
8F00 - BFFF	48896 - 49151	Processor zero-page storage for hypervisor or operating system

The stack is used for holding the return address of function calls. The zero-page storage is typically used for holding variables and other short-term storage, as is customary on the 6502.

Hypervisor Virtualisation Control Registers

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D640	54848					REGA			
D641	54849					REGX			
D643	54851					REGZ			
D644	54852					REGB			
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	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
- **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
- **RSVD** RESERVED
- **SPH** Hypervisor SPH register storage
- **SPL** Hypervisor SPL register storage
- **UARTDATA** (write) Hypervisor write serial output to UART monitor
- **VFLOP** 1=Virtualise SD/Floppy0 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 16KB 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 KERNEL Development Template
// Each function of the KERNEL is a no-args function
// The functions are placed in the SYSCALLS 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 = 0xd800;
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 SYSCALL handlers that just display a character on the screen
void syscall1() {
    *(SCREEN+79) = '>';
}

void syscall2() {
    *(SCREEN+78) = '<';
}

// Now we select the SYSCALL segment to hold the SYSCALL/trap entry point table.
#pragma data+seg(Syscall)

// The structure of each entry point is JMP <handler address> + NOP.
// We have a char (xjmp) to hold the opcode for the JMP instruction,
// and then put the address of the SYSCALL/trap handler in the next
// two points as a pointer, and end with the NOP instruction opcode.
```

```

struct SysCall {
    char xjmp;          // Holds $4C, the JMP $nnnn opcode
    void(*) syscall;   // Holds handler address, will be the target of the JMP
    char xnop;          // Holds $EA, the NOP opcode
};

// To save writing 0x4C and 0xEA all the time, we define them as constants
const char JMP = 0x4c;
const char NOP = 0xea;

// Now we can have a nice table of up to 64 SYSCALL handlers expressed
// in a fairly readable and easy format.
// Each line is an instance of the struct SysCall from above, with the JMP
// opcode value, the address of the handler routine and the NOP opcode value.
export struct SysCall[] SYSCALLS = {
    { JMP, &syscall1, NOP },
    { JMP, &syscall2, NOP }
};

// In this example we had only two SYSCALLs defined, so rather than having
// another 62 lines, we can just ask KickC to make the TRAP table begin
// at the next multiple of $100, i.e., at $8100.
export align(0x100) struct SysCall[] SYSCALLs+RESET = {
    { JMP, &main, NOP }
};

```

If you save the first listing into a file called mega65hyper.ld, and the second into a file called mega65hyper.kc, you can then compile them using KickC with a command like:

```
kickc -a mega65hyper
```

It will then produce a file called mega65hyper.bin, which you can then try out on your MEGA65, or run in the XMega65 emulator with a command like:

```
xmega65 -kickup mega65hyper.bin
```

H

APPENDIX

45GS02 & 6502 Instruction Sets

- **Introduction**
- **Stack Operations**
- **Addressing Modes**
- **6502 Instruction Set**
- **4510 Instruction Set**
- **45GS02 Compound Instructions**

INTRODUCTION

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.

WARNING: This feature is incomplete and untested. Most undocumented 6502 opcodes do not operate correctly when the 6502 personality is enabled.

When in 4510 compatibility mode, the 45GS02 also supports a number of extensions through *compound instructions*. These work by prefixing the desired instruction's opcode with one or more *prefix bytes*, which represent sequences of instructions that should not normally occur. For example, two **NEG** instructions in a row acts as a prefix to tell the 45GS02 that the following instruction will operate on 32 bits of data, instead of the usual 8 bits of data. This means that a 45GS02 instruction stream can be readily decoded or disassembled, without needing to set special instruction length flags, as is the case with the 65816 family of microprocessors. The trade-off is increased execution time, as the 45GS02 must skip over the prefix bytes.

The remainder of this chapter introduces the addressing modes, instructions, opcodes and instruction timing data of the 45GS02, beginning with 6502 compatibility mode, before moving on to 4510 compatibility mode, and the 45GS02 extensions.

STACK OPERATIONS

The stack is an area of memory where you can push data onto or fetch (pop) the latest piece of data from it. Every stack operation that puts data to the stack (push) also changes the Stack Pointer (SP) downwards (the stack starts at the top of the area and grows down), and every stack operation that takes data from the stack (pop) will change the SP upwards.

So if you find something like

STACK \leftarrow VALUE

implies that VALUE is pushed onto the STACK and SP is reduced by the number of bytes VALUE is big. When pushing more than one byte, the MSB is pushed first followed by the LSB.

And in the opposite direction a operation like

MEMORY or REGISTER \leftarrow STACK

will pop data from the STACK and increment SP for each byte removed.

ADDRESSING MODES

The 45GS02 supports 34 different addressing modes, which are explained below. Many of these are very similar to one another, being variations of the normal 6502 or 65CE02 addressing modes, except that they accept either 32-bit pointers, operate on 32-bits of data, or both.

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.

Q Pseudo Register

In this mode, the Q Pseudo Register is the operand. This is typically used to shift, rotate or modify the value of the Q Pseudo Register in some way. For example, `ASLQ` shifts the value in the Q Pseudo Register left one bit.

Remember that the Q Pseudo Register is simply the A, X, Y and Z registers acting together as a virtual 32-bit register, where A contains the least significant bits, and Z the most significant bits. If you modify Q, you will modify the true registers, and similarly, if you modify a true register, this will change the respective part of the Q register.

There are some cases where using a Q mode instruction can be helpful for operating on the four true registers, for example, being able to quickly load or store all four registers.

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 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 Quad Mode

This mode is identical to Base-Page Mode, except that it reads a 32-bit word starting at the specified address.

The argument is encoded as a single byte that immediately follows the instruction opcode. For example,

```
LDD $12
```

would read the value stored in locations \$12 - \$15 in the Base-Page, and put them into the Q Pseudo Register. The instruction byte stream for this would be \$42 \$42 \$85 \$12. Note that this is the same as for the Base-Page (Zero-Page) Mode, with the addition of the two \$42 prefix bytes. Opcode \$42 is normally NEG (negate the value in the A register). When executed twice in a row, this returns the A value to its original value. The 45GS02 processor has special logic to recognises this sequence, so that it knows to execute the next instruction using the Q Pseudo Register for that instruction.

See the note on page [H-5](#) for more information about Base-Page and Zero-Page.

Base-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.

The argument is encoded as a single byte that immediately follows the instruction opcode. For example,

```
LDA $12,X
```

would read the value stored in location (\$12 + X) in the Base-Page, and put it into the A register. The instruction byte stream for this would be \$B5 \$12.

See the note on page [H-5](#) for more information about Base-Page and Zero-Page.

Base-Page Quad X-Indexed Mode

This mode is identical to Base-Page Quad 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 Quad Mode.

The argument is encoded as a single byte that immediately follows the instruction opcode. For example,

```
DEQ $12,X
```

would increment the 32-bit word stored at (\$12 + X) through to (\$15 + X) in the Base-Page, and put it into the X register. The instruction byte stream for this would be \$42 \$42 \$D6 \$12.

Note that LDQ is not available in this addressing mode.

See the note on page [H-5](#) for more information about Base-Page and Zero-Page. See the note on page [H-6](#) for more information on Quad Mode instructions.

Base-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.

The argument is encoded as a single byte that immediately follows the instruction opcode. For example,

```
LDX $12,Y
```

would read the value stored in location (\$12 + Y) in the Base-Page, and put it into the X register. The instruction byte stream for this would be \$B6 \$12.

See the note on page [H-5](#) for more information about Base-Page and Zero-Page.

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 Quad 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

LDQ \$1234

would read the memory locations \$1234 - \$1237, and place the read values into the Q Pseudo Register. This would be encoded as \$42 \$42 \$AD \$34 \$12.

See the note on page [H-6](#) for more information on Quad Mode instructions.

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. For example, the instruction

LDA \$1234,X

would read the memory location (\$1234 + X), and place the value read from there into the A Register. This would be encoded as \$BD \$34 \$12.

Absolute Quad X-Indexed Mode

This mode is identical to Absolute Quad 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 Quad Mode.

For example, the instruction

```
ROLQ $1234,X
```

would rotate left the 32-bit value at memory locations (\$1234+X) - (\$1237+X), and write the result back to these same memory locations. This would be encoded as \$42 \$42 \$3E \$34 \$12.

See the note on page [H-6](#) for more information on Quad Mode instructions.

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. For example, the instruction

```
LDA $1234,Y
```

would read the memory location (\$1234 + Y), and place the value read from there into the A Register. This would be encoded as \$B9 \$34 \$12.

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, and thus this instruction would be encoded as \$6C \$34 \$12.

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

```
JMP ($1234,X)
```

would jump to address \$5678. The encoding for this addressing mode is identical to Absolute Mode, and thus this instruction would be encoded as \$7C \$34 \$12.

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.

For example, if the X Register contains the value \$04, and the memory locations \$16 and \$17 in the current Base-Page contained \$34 and \$12, respectively, then

```
LDA ($12,X)
```

would read the contents of memory location \$1234, and store the result in the A register. This instruction would be encoded as \$A1 \$12.

See the note on page [H-5](#) for more information about Base-Page and Zero-Page.

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 used 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.

For example, if the Y Register contains the value \$04, and the memory locations \$12 and \$13 in the current Base-Page contained \$78 and \$56, respectively, then

```
LDA ($12),Y
```

would read the contents of memory location \$567C (i.e., \$5678 + Y), and store the result in the A register. This instruction would be encoded as \$B1 \$12.

See the note on page [H-5](#) for more information about Base-Page and Zero-Page.

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 used 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.

For example, if the Z Register contains the value \$04, and the memory locations \$12 and \$13 in the current Base-Page contained \$78 and \$56, respectively, then

```
LDA ($12),Z
```

would read the contents of memory location \$567C (i.e., \$5678 + Z), and store the result in the A register. This instruction would be encoded as \$B2 \$12.

That is, it is equivalent to the Base-Page Indirect Y-Indexed Mode, but using the Z register instead of the Y register to calculate the offset.

See the note on page [H-5](#) for more information about Base-Page and Zero-Page.

Base-Page Quad Indirect Z-Indexed Mode

This addressing mode is identical to the Base-Page Indirect Z-Indexed Mode, except that 32-bits of data are operated on. The encoding for this addressing mode is identical to Base-Page Mode, except that it is prefixed by \$42, \$42. For example, if the Z Register contains the value \$04, and the memory locations \$20, and \$21 in the current Base-Page contained \$AB and \$CD, respectively, then

```
LDQ ($12),Z
```

would read the contents of memory location \$ABD1 (i.e., \$ABCD + Y) - \$ABD4 and store the result in the Q Pseudo Register. This instruction would be encoded as \$42 \$42 \$B2 \$12.

Currently the only instruction that offers this mode is LDQ.

See the note on page [H-5](#) for more information about Base-Page and Zero-Page. See the note on page [H-6](#) for more information on Quad Mode instructions.

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, \$30, \$FD, \$0F, representing the 32-bit address \$FFD3020, i.e., the VIC-IV border colour register's natural address, and the Z index contained the value \$01, the following instruction sequence would change the screen colour to blue, because the screen colour register is at \$FFD3021, i.e., \$FFD3020 + Z:

```
LDA #$06  
LDZ #$01  
STA [$12],Z
```

See the note on page [H-5](#) for more information about Base-Page and Zero-Page.

32-bit Base-Page (Zero-Page) Indirect Quad Z-Indexed Mode

This addressing mode is identical to the 32-bit Base-Page Indirect Z-Indexed Mode, except that it operates on 32-bits of data at the 32-bit address formed by the argument, in comparison to 32-bit Base-Page Indirect Z-Indexed Mode which operates on only 8 bits of data. The encoding of this addressing mode is \$42, \$42, \$EA, followed by the natural 6502 opcode for the instruction being performed.

It is also important to note that most of the time Z is actually *not* added to the pointer, as it is part of the Q Pseudo Register. Only LDQ (\$nn),Z will add Z. For example,

```
LDQ [$12],Z
```

would read the memory locations at $\$12 + Z$ through $\$15 + Z$ from the Base-Page, and use those values to form the 32-bit address from which to load the Q Pseudo Register. This instruction would be encoded as **$\$42\ \$42\ \$EA\ \$B2\ \$12$** .

See the note on page [H-5](#) for more information about Base-Page and Zero-Page. See the note on page [H-6](#) for more information on Quad Mode instructions. See the note on page [H-12](#) for more information on 32-bit Base-Page Indirect addressing.

Stack Relative Indirect, Y-Indexed

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.

For example,

```
LDA ($12,SP),Y
```

This would use the contents of memory at the current stack pointer plus $\$12$ to compute the address of the pointer. This pointer would then have the Y-Register value added to it to obtain the final address to read from, and to store the value into the Accumulator. The instruction byte stream for this instruction would be **$\$E2\ \12** .

If the Stack Pointer currently pointed to $\$01E0$, then the pointer address would be read from addresses $\$1F2$ and $\$1F3$, i.e., the two bytes at $\$1E0 + \12 and $\$1E0 + \13 . If locations $\$1F2$ and $\$1F3$ contained $\$78$ and $\$56$ respectively, and the Y-Register contained the value $\$34$, then the final memory location that would be read would be $\$5678 + \$34 = \$56AC$, and the contents of that memory location would be read into the Accumulator.

Relative Addressing Mode

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

For example,

```
BNE $2003
```

would jump to \$2003, if the Z flag of the processor was not set. If this instruction were located at address \$2000, it would be encoded as \$D0 \$01, i.e., branching to +1 bytes after the PC. Branch offsets greater than \$7F branch backwards, with \$FD branching to the byte immediately preceding the branch instruction, and lower values branching progressively further back. In this way, a branch can effectively be made between -125 and +127 bytes from the opcode byte of the branch instruction. For longer branches, the 45GS02 supports Relative Word Addressing Mode, where the offset is encoded using 2 bytes instead of 1.

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.

For example,

```
BNE $3000
```

would jump to \$3000 if the Z flag of the process was not set. If this instruction were located at address \$2002, it would be encoded as \$D3 \$FC \$0F, i.e., branching to +\$FFC = 4,092 bytes following the second byte of the instruction. The fact that the instruction is 3 bytes long is ignored in this calculation.

6502 INSTRUCTION SET

NOTE: The mechanisms for switching from 4510 to 6502 CPU personality have yet to be finalised.

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.

NOTE: The unintended instructions are currently unimplemented, and are guaranteed not 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.

Opcode Table

The Opcode Table lists all possible opcodes, their size, cycles and addressing mode in a concise format.

A cell with a light red background signifies an **unintended** instruction.

	\$x0	\$x1
\$0x	size OPC mode	size OPC mode

The letters attached to the cycle count have the following meaning:

	Meaning
b	Add one cycle if branch is taken. Add one more cycle if branch taken crosses a page boundary.
p	Add one cycle if indexing crosses a page boundary.

Opcode Table 6502

	\$x0	\$x1	\$x2	\$x3	\$x4	\$x5	\$x6	\$x7
\$0x	1 BRK imp	7 2 ORA bpX	6 1 KIL imp	9 2 SLO bpX	8 2 NOP bp	3 2 ORA bp	3 2 ASL bp	5 2 SLO bp
\$1x	2 BPL rel	2 5p ORA ibpY	1 9 KIL imp	2 8 SLO ibpY	2 4 NOP bpX	4p 2 ORA bpX	6 2 ASL bpX	6 2 SLO bpX
\$2x	3 JSR abs	6 2 AND bpX	6 1 KIL imp	9 2 RLA bpX	8 2 BIT bp	3 2 AND bp	3 2 ROL bp	5 2 RLA bp
\$3x	2 BMI rel	2 2b 5p AND ibpY	1 9 KIL imp	9 2 RLA ibpY	8 2 NOP bpX	4 2 AND bpX	4 2 ROL bpX	6 2 RLA bpX
\$4x	1 RTI imp	6 2 EOR bpX	6 1 KIL imp	9 2 SRE bpX	8 2 NOP bp	3 2 EOR bp	5 2 LSR bp	5 2 SRE bp
\$5x	2 BVC rel	2 2b 5p EOR ibpY	1 9 KIL imp	2 8 SRE ibpY	2 4 NOP bpX	4 2 EOR bpX	6 2 LSR bpX	6 2 SRE bpX
\$6x	1 RTS imp	6 2 ADC bpX	6 1 KIL imp	9 2 RRA bpX	8 2 NOP bp	3 2 ADC bp	3 2 ROR bp	5 2 RRA bp
\$7x	2 BVS rel	2 2b 5p ADC ibpY	1 9 KIL imp	2 8 RRA ibpY	2 4 NOP bpX	4 2 ADC bpX	6 2 ROR bpX	6 2 RRA bpX
\$8x	2 NOP imm	2 2 STA bpX	6 2 NOP imm	2 6 SAX bpX	2 3 STY bp	3 2 STA bp	3 2 STX bp	3 2 SAX bp
\$9x	2 BCC rel	2 2b STA ibpY	6 1 KIL imp	9 2 SHA ibpY	6 2 STY bpX	4 2 STA bpX	4 2 STX bpY	4 2 SAX bpY
\$Ax	2 LDY imm	2 2 LDA bpX	6 2 LDX imm	2 6 LAX bpX	2 3 LDY bp	3 2 LDA bp	3 2 LDX bp	3 2 LAX bp
\$Bx	2 BCS rel	2 2b 5p LDA ibpY	1 9 KIL imp	2 5p LAX ibpY	2 4 LDY bpX	4 2 LDA bpX	4 2 LDX bpY	4 2 LAX bpY
\$Cx	2 CPY imm	2 2 CMP bpX	6 2 NOP imm	2 8 DCP bpX	2 3 CPY bp	3 2 CMP bp	5 2 DEC bp	5 2 DCP bp
\$Dx	2 BNE rel	2 2b 5p CMP ibpY	1 9 KIL imp	2 8 DCP ibpY	2 4 NOP bpX	4 2 CMP bpX	6 2 DEC bpX	6 2 DCP bpX
\$Ex	2 CPX imm	2 2 SBC bpX	6 2 NOP imm	2 8 ISC bpX	2 3 CPX bp	3 2 SBC bp	5 2 INC bp	5 2 ISC bp
\$Fx	2 BEQ rel	2 2b 5p SBC ibpY	1 9 KIL imp	2 8 ISC ibpY	2 4 NOP bpX	4 2 SBC bpX	6 2 INC bpX	6 2 ISC bpX

Opcode Table 6502

\$x8	\$x9	\$xA	\$xB	\$xC	\$xD	\$xE	\$xF	
1 PHP imp	3 ORA imm	2 1 ASL acc	2 2 ANC imm	2 3 NOP abs	4 3 ORA abs	3 3 ASL abs	6 3 SLO abs	\$0x
1 CLC imp	2 3 ORA absY	4p 1 NOP imp	2 3 SLO absY	7 3 NOP absX	4p 3 ORA absX	3 3 ASL absX	7 3 SLO absX	\$1x
1 PLP imp	4 2 AND imm	2 1 ROL acc	2 2 ANC imm	2 3 BIT abs	4 3 AND abs	6 3 ROL abs	6 3 RLA abs	\$2x
1 SEC imp	2 3 4p AND absY	1 NOP imp	2 3 RLA absY	7 3 NOP absX	4p 3 AND absX	7 3 ROL absX	7 3 RLA absX	\$3x
1 PHA imp	3 2 EOR imm	2 1 LSR acc	2 2 ALR imm	2 3 JMP abs	3 3 EOR abs	4 3 LSR abs	6 3 SRE abs	\$4x
1 CLI imp	2 3 4p EOR absY	1 NOP imp	2 3 SRE absY	7 3 NOP absX	4p 3 EOR absX	7 3 LSR absX	7 3 SRE absX	\$5x
1 PLA imp	4 2 ADC imm	2 1 ROR acc	2 2 ARR imm	2 3 JMP ind	5 3 ADC abs	4 3 ROR abs	6 3 RRA abs	\$6x
1 SEI imp	2 3 4p ADC absY	1 NOP imp	2 3 RRA absY	7 3 NOP absX	4p 3 ADC absX	7 3 ROR absX	7 3 RRA absX	\$7x
1 DEY imp	2 2 NOP imm	2 1 TXA imp	2 2 XAA imm	2 3 STY abs	4 3 STA abs	4 3 STX abs	4 3 SAX abs	\$8x
1 TYA imp	2 3 5 STA absY	1 TXS imp	2 3 TAS absY	5 3 SHY absX	5 3 STA absX	5 3 SHX absY	5 3 SHA absY	\$9x
1 TAY imp	2 2 LDA imm	2 1 TAX imp	2 2 LAX imm	2 3 LDY abs	4 3 LDA abs	4 3 LDX abs	4 3 LAX abs	\$Ax
1 CLV imp	2 3 4p LDA absY	1 TSX imp	2 3 4p LAS absY	4p 3 LDY absX	4p 3 LDA absX	4p 3 LDX absY	4p 3 LAX absY	\$Bx
1 INY imp	2 2 2 CMP imm	2 1 DEX imp	2 2 SBX imm	2 3 CPY abs	4 3 CMP abs	6 3 DEC abs	6 3 DCP abs	\$Cx
1 CLD imp	2 3 4p CMP absY	1 NOP imp	2 3 DCP absY	7 3 NOP absX	4p 3 CMP absX	7 3 DEC absX	7 3 DCP absX	\$Dx
1 INX imp	2 2 2 SBC imm	2 1 NOP imp	2 2 SBC imm	2 3 CPX abs	4 3 SBC abs	6 3 INC abs	6 3 ISC abs	\$Ex
1 SED imp	2 3 4p SBC absY	1 NOP imp	2 3 ISC absY	7 3 NOP absX	4p 3 SBC absX	7 3 INC absX	7 3 ISC absX	\$Fx

ADC

This instruction adds the argument and the Carry Flag to the contents of the Accumulator Register. 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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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		N	Z	I	C	D	V	E
		+	+	.	+	.	+	.
Addressing Mode	Assembly	Code	Bytes	Bytes	Cycles			
(indirect,X)	ADC (\$nn,X)	61	2	2	6			
zero-page	ADC \$nn	65	2	2	3			
immediate 8bit	ADC #\$nn	69	2	2	2			
absolute	ADC \$nnnn	6D	3	3	4			
(indirect),Y	ADC (\$nn),Y	71	2	2	5	p		
zero-page,X	ADC \$nn,X	75	2	2	4			
absolute,Y	ADC \$nnnn,Y	79	3	3	4	p		
absolute,X	ADC \$nnnn,X	7D	3	3	4	p		

p Add one cycle if indexing crosses a page boundary.

ALR [unintended]

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The C flag will be set to bit 0 of the AND result, prior to being shifted.

ALR : Binary AND and Logical Shift Right		6502
A \leftarrow A AND M, C \leftarrow A(0), A \leftarrow A \gg 1, A(7) \leftarrow 0		
		N Z I C D V E
		+ + . + . . .
Addressing Mode	Assembly	Code
immediate 8bit	ALR #\$nn	4B
		Bytes Cycles
		2 2

ANC [unintended]

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The C flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

ANC : Binary AND, and Set Carry		6502
A \leftarrow A AND M, C \leftarrow A(7)		
		N Z I C D V E
		+ +
Addressing Mode	Assembly	Code
immediate 8bit	ANC #\$nn	0B
immediate 8bit	ANC #\$nn	2B
		Bytes Cycles
		2 2

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

AND : Binary AND		6502						
A ← A AND M		N	Z	I	C	D	V	E
		+	+
Addressing Mode	Assembly	Code	Bytes	Bytes	Cycles			
(indirect,X)	AND (\$nn,X)	21	2	6				
zero-page	AND \$nn	25	2	3				
immediate 8bit	AND #\$nn	29	2	2				
absolute	AND \$nnnn	2D	3	4				
(indirect),Y	AND (\$nn),Y	31	2	5				p
zero-page,X	AND \$nn,X	35	2	4				
absolute,Y	AND \$nnnn,Y	39	3	4				p
absolute,X	AND \$nnnn,X	3D	3	4				p

p Add one cycle if indexing crosses a page boundary.

ARR [unintended]

This instruction shifts the Accumulator one bit right after performing a binary AND of the Accumulator and the immediate mode argument M . Bit 7 is exchanged with the carry.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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 AND M, A \leftarrow A \gg 1, C \leftrightarrow A(7)				
Addressing Mode	Assembly	Code	Bytes	Cycles
immediate 8bit	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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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 Memory or Accumulator		6502		
A \leftarrow A \ll 1 or M \leftarrow M \ll 1				
Addressing Mode	Assembly	Code	Bytes	Cycles
zero-page	ASL \$nn	06	2	5
accumulator	ASL A	0A	1	2
absolute	ASL \$nnnn	0E	3	6
zero-page,X	ASL \$nn,X	16	2	6
absolute,X	ASL \$nnnn,X	1E	3	7

BCC

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

BCC : Branch on Carry Flag Clear	6502
$C=0 \implies PC \leftarrow PC + R8$	
	N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BCC \$rr	90	2	2 ^b

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	6502
$C=1 \implies PC \leftarrow PC + R8$	
	N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BCS \$rr	B0	2	2 ^b

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. BEW stands for branch if equal, because a CMP will result in the zero flag being set if the operands are equal.

BEQ : Branch on Zero Flag Set	6502
$Z=1 \implies PC \leftarrow PC + R8$	
	N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BEQ \$rr	F0	2	2 ^b

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, otherwise it will be cleared.
- The V flag will be set if the bit 6 of the memory location is set, otherwise it will be cleared.
- The Z flag will be set if the result of A AND M is zero, otherwise it will be cleared.

BIT : Perform Bit Test		6502		
Addressing Mode	Assembly	Code	Bytes	Cycles
zero-page	BIT \$nn	24	2	3
absolute	BIT \$nnnn	2C	3	4

BMI

This instruction branches to the indicated address if the Negative Flag is set. BMI stands for branch on minus.

BMI : Branch on Negative Flag Set		6502		
Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BMI \$rr	30	2	2 ^b

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 stands for Branch if not equal, because a CMP will result in the zero flag being cleared if the operants are not equal.

BNE : Branch on Zero Flag Clear		6502
Z=0 \Rightarrow PC \leftarrow PC + R8		N Z I C D V E
	
Addressing Mode	Assembly	Code
relative	BNE \$rr	D0
		2 2 ^b

^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 stands for branch on plus.

BPL : Branch on Negative Flag Clear		6502
N=0 \Rightarrow PC \leftarrow PC + R8		N Z I C D V E
	
Addressing Mode	Assembly	Code
relative	BPL \$rr	10
		2 2 ^b

^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		6502		
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	BRK	00	1	7

BVC

This instruction branches to the indicated address if the Overflow (V) Flag is clear.

BVC : Branch on Overflow Flag Clear		6502		
Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BVC \$rr	50	2	2 ^b

^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	6502
$V=1 \implies PC \leftarrow PC + R8$	
	N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BVS \$rr	70	2	2 ^b

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	6502			
$C \leftarrow 0$				
	N Z I C D V E . . . + . .			
Addressing Mode	Assembly	Code	Bytes	Cycles
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.

CLD : Clear Decimal Flag	6502			
$D \leftarrow 0$				
	N Z I C D V E . . . + . .			
Addressing Mode	Assembly	Code	Bytes	Cycles
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.

CLI : Clear Interrupt Disable Flag		6502		
I ← 0		N Z I C D V E		
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	CLI	58	1	2

CLV

This instruction clears the Overflow Flag.

Side effects

- The V flag is cleared.

CLV : Clear Overflow Flag		6502		
V ← 0		N Z I C D V E		
Addressing Mode	Assembly	Code	Bytes	Cycles
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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The Z flag will be set if the result of $A - M$ is zero, otherwise it will be cleared.

CMP : Compare Accumulator			6502	
Addressing Mode	Assembly	Code	Bytes	Cycles
(indirect,X)	CMP (\$nn,X)	C1	2	6
zero-page	CMP \$nn	C5	2	3
immediate 8bit	CMP #\$nn	C9	2	2
absolute	CMP \$nnnn	CD	3	4
(indirect),Y	CMP (\$nn),Y	D1	2	5 ^p
zero-page,X	CMP \$nn,X	D5	2	4
absolute,Y	CMP \$nnnn,Y	D9	3	4 ^p
absolute,X	CMP \$nnnn,X	DD	3	4 ^p

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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The Z flag will be set if the result of $X - M$ is zero, otherwise it will be cleared.

CPX : Compare X Register	6502			
$N, C, Z \leftarrow [X - M]$				
	N Z I C D V E			
	+ + . + . .			
Addressing Mode	Assembly	Code	Bytes	Cycles
immediate 8bit	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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The Z flag will be set if the result of $Y - M$ is zero, otherwise it will be cleared.

CPY : Compare Y Register	6502			
$N, C, Z \leftarrow [Y - M]$				
	N Z I C D V E			
	+ + . + . .			
Addressing Mode	Assembly	Code	Bytes	Cycles
immediate 8bit	CPY #\$nn	C0	2	2
zero-page	CPY \$nn	C4	2	3
absolute	CPY \$nnnn	CC	3	4

DCP [unintended]

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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The Z flag will be set if the result of A – M is zero, otherwise it will be cleared.

DCP : Decrement and Compare Accumulator		6502						
$M \leftarrow M - 1, N,C,Z \Leftarrow [A - M]$		N	Z	I	C	D	V	E
Addressing Mode	Assembly	Code	Bytes	Cycles				
(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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

DEC : Decrement		6502		
$A \leftarrow A - 1$ or $M \leftarrow M - 1$				
	N Z I C D V E			
	+ +			
Addressing Mode	Assembly	Code	Bytes	Cycles
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

DEX : Decrement X Register		6502		
$X \leftarrow X - 1$				
	N Z I C D V E			
	+ +			
Addressing Mode	Assembly	Code	Bytes	Cycles
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

DEY : Decrement Y Register	6502
$Y \leftarrow Y - 1$	
	N Z I C D V E + +
Addressing Mode	Assembly
implied	DEY
	Code
	88
	Bytes
	1
	Cycles
	2

eor

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, but not both.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

EOR : Binary Exclusive OR	6502
$A \leftarrow A \text{ XOR } M$	
	N Z I C D V E + +
Addressing Mode	Assembly
(indirect,X)	EOR (\$nn,X)
zero-page	EOR \$nn
immediate 8bit	EOR #\$nn
absolute	EOR \$nnnn
(indirect),Y	EOR (\$nn),Y
zero-page,X	EOR \$nn,X
absolute,Y	EOR \$nnnn,Y
absolute,X	EOR \$nnnn,X
	Code
	41
	45
	49
	4D
	51
	55
	59
	5D
	Bytes
	2
	2
	2
	3
	2
	2
	3
	3
	3
	Cycles
	6
	3
	2
	4
	5 ^p
	4
	4 ^p
	4 ^p

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

INC : Increment 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		
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

INX : Increment X Register		6502				
X \leftarrow X + 1		N Z I C D V E				
		+ +				
Addressing Mode	Assembly	Code	Bytes	Cycles		
implied	INX	E8	1	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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

INY : Increment Y Register		6502
Y \leftarrow Y + 1		
		N Z I C D V E
		+ +
Addressing Mode	Assembly	Code
implied	INY	C8
		Bytes Cycles
		1 2

ISC [unintended]

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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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, otherwise it will be cleared.

ISC : Increment Memory, Subtract With Carry**6502** $M \leftarrow M + 1, A \leftarrow -M - 1 + C$

N	Z	I	C	D	V	E
+	+	.	+	.	+	.

Addressing Mode	Assembly	Code	Bytes	Cycles
(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 Program Counter (PC) Register to the address indicated by the instruction, causing execution to continue from that address.

JMP : Jump to Address**6502** $PC \leftarrow M2:M1$

N	Z	I	C	D	V	E
.

Addressing Mode	Assembly	Code	Bytes	Cycles
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 Program 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 Program Counter (PC) register.

JSR : Jump to Sub-Routine	6502			
PC ← M2:M1, STACK ← PCH:PCL				
N Z I C D V E				
Addressing Mode	Assembly	Code	Bytes	Cycles
absolute	JSR \$nnnn	20	3	6

KIL [unintended]

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, 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 program to be aborted or crash.

KIL : Lock-up 6502 Processor	6502			
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 [unintended]

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- A feeling of hollow satisfaction, when you actually discover a useful purpose for this instruction.

LAS : Set A, X and SPL Register With Useless Value		6502			
Addressing Mode	Assembly	Code	Bytes	Cycles	
absolute,Y	LAS \$nnnn,Y	BB	3	4	^p

p Add one cycle if indexing crosses a page boundary.

LAX [unintended]

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

LAX : Load Accumulator and X Registers**6502**A, X \leftarrow M

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
(indirect,X)	LAX (\$nn,X)	A3	2	6	
zero-page	LAX \$nn	A7	2	3	
immediate 8bit	LAX #\$nn	AB	2	2	
absolute	LAX \$nnnn	AF	3	4	
(indirect),Y	LAX (\$nn),Y	B3	2	5	<i>p</i>
zero-page,Y	LAX \$nn,Y	B7	2	4	
absolute,Y	LAX \$nnnn,Y	BF	3	4	<i>p</i>

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

LDA : Load Accumulator**6502**A \leftarrow M

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
(indirect,X)	LDA (\$nn,X)	A1	2	6	
zero-page	LDA \$nn	A5	2	3	
immediate 8bit	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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

LDX : Load X Register**6502**X \leftarrow M

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
immediate 8bit	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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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 8bit	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	4	<i>p</i>	

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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
$A \leftarrow A \gg 1, C \leftarrow A(0)$ or $M \leftarrow M \gg 1, C \leftarrow M(0)$						
						N Z I C D V E
						+
Addressing Mode	Assembly	Code	Bytes	Cycles		
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.

NOP : No-Operation (some are unintended opcodes)

6502

N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles	
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	<i>p</i>
zero-page,X	NOP \$nn,X	34	2	4	
implied	NOP	3A	1	2	
absolute,X	NOP \$nnnn,X	3C	3	4	<i>p</i>
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	<i>p</i>
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	<i>p</i>
immediate 8bit	NOP #\$nn	80	2	2	
immediate 8bit	NOP #\$nn	82	2	2	
immediate 8bit	NOP #\$nn	89	2	2	
immediate 8bit	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	<i>p</i>
immediate 8bit	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	<i>p</i>

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

ORA : Binary OR		6502				
A ← A OR M		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 8bit	ORA #\$nn	09	2	2		
absolute	ORA \$nnnn	0D	3	4		
(indirect),Y	ORA (\$nn),Y	11	2	5	p	
zero-page,X	ORA \$nn,X	15	2	4	p	
absolute,Y	ORA \$nnnn,Y	19	3	4	p	
absolute,X	ORA \$nnnn,X	1D	3	4	p	

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 ← A, SP ← 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.

PHP : Push Processor Flags onto the Stack		6502
STACK \leftarrow P, SP \leftarrow SP - 1		
		N Z I C D V E
Addressing Mode	Assembly	Code
implied	PHP	08
Bytes	Cycles	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.

PLA : Pull Accumulator Register from the Stack		6502
A \leftarrow STACK, SP \leftarrow SP + 1		
		N Z I C D V E
Addressing Mode	Assembly	Code
implied	PLA	68
Bytes	Cycles	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)

PLP : Pull Processor Flags from the Stack		6502
A \leftarrow STACK, SP \leftarrow SP + 1		
		N Z I C D V E
Addressing Mode	Assembly	Code
implied	PLP	28
Bytes	Cycles	1 4

RLA [unintended]

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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				
$M \leftarrow M \ll 1, M(0) \leftarrow C, C \leftarrow M(7), A \leftarrow A \text{ AND } M$						
Addressing Mode	Assembly	Code	N	Z	I	C
(indirect,X)	RLA (\$nn,X)	23	+	+	.	+
zero-page	RLA \$nn	27	2			5
absolute	RLA \$nnnn	2F		3		6
(indirect),Y	RLA (\$nn),Y	33	2			8
zero-page,X	RLA \$nn,X	37	2			6
absolute,Y	RLA \$nnnn,Y	3B		3		7
absolute,X	RLA \$nnnn,X	3F		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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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

 $M \leftarrow M \ll 1, C \leftarrow M(7), M(0) \leftarrow C$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles
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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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

 $M \leftarrow M \gg 1, C \leftarrow M(0), M(7) \leftarrow C$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles
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 [unintended]

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The C flag will be set if the addition results in an overflow in the Accumulator.

RRA : Rotate Right Memory, and Add to Accumulator		6502		
		M \leftarrow M \gg 1, C \leftarrow M(0), M(7) \leftarrow C, A \leftarrow A + M		
		N Z I C D V E		
		+ + . + . . .		
Addressing Mode	Assembly	Code	Bytes	Cycles
(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 Program Counter (PC) register from the stack, allowing an interrupted program 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.

RTI : Return From Interrupt	6502			
$P \leftarrow \text{STACK}$, $\text{PC} \leftarrow \text{STACK}$				
N Z I C D V E + + + + + +				
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	RTI	40	1	6

RTS

This instruction adds an optional argument to the Stack Pointer (SP) Register, and then pops the Program Counter (PC) register from the stack, allowing a routine to return to its caller.

RTS : Return From Subroutine	6502			
$PC \leftarrow \text{STACK}$ or $PC \leftarrow \text{STACK} + M$, $SP \leftarrow SP - 2$				
N Z I C D V E				
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	RTS	60	1	6

SAX [unintended]

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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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, otherwise it will be cleared.

$$X \leftarrow (A \text{ AND } X) - M$$

N	Z	I	C	D	V	E
+	+	.	+	.	+	.

Addressing Mode	Assembly	Code	Bytes	Cycles
(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: 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 of $A - M$ is negative, i.e. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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, otherwise it will be cleared.

SBC : Subtract With Carry

6502

$$A \leftarrow -M - 1 + C$$

N	Z	I	C	D	V	E
+	+	.	+	.	+	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
(indirect,X)	SBC (\$nn,X)	E1	2	6	
zero-page	SBC \$nn	E5	2	3	
immediate 8bit	SBC #\$nn	E9	2	2	
immediate 8bit	SBC #\$nn	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 [unintended]

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The C flag will be set if the result is zero or positive, otherwise it will be cleared.

SBX : AND and Subtract

6502

$$X \leftarrow (A \text{ AND } X) - M$$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles
immediate 8bit	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	Cycles
implied	SEC	38	1	2

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	Cycles
implied	SED	F8	1	2

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.

SEI : Set Interrupt Disable Flag		6502		
I ← 1				
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	SEI	78	1	2

SHA [unintended]

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.

SHA : Store binary AND of A, X and 3rd Instruction Byte		6502		
M ← A AND X AND B3				
Addressing Mode	Assembly	Code	Bytes	Cycles
(indirect),Y	SHA (\$nn),Y	93	2	6
absolute,Y	SHA \$nnnn,Y	9F	3	5

SHX [unintended]

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.

SHX : Store Binary AND of X Register and 3rd Instruction Byte		6502		
M ← X AND B3				
Addressing Mode	Assembly	Code	Bytes	Cycles
absolute,Y	SHX \$nnnn,Y	9E	3	5

SHY [unintended]

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.

SHY : Store Binary AND of Y Register and 3rd Instruction Byte		6502		
M ← Y AND B3		N Z I C D V E		
Addressing Mode	Assembly	Code	Bytes	Cycles
absolute,X	SHY \$nnnn,X	9C	3	5

SLO [unintended]

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the Accumulator contains \$00 after the instruction has completed, otherwise 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

 $C \leftarrow M(7), M \leftarrow M \ll 1, A \leftarrow A \text{ OR } M$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles
(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 [unintended]

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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

 $C \leftarrow M(0), M \leftarrow M \gg 1, A \leftarrow A \text{ XOR } M$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles
(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.

STY : Store Y Register		6502		
$M \leftarrow Y$				
		N Z I C D V E		
			
Addressing Mode	Assembly	Code	Bytes	Cycles
zero-page	STY \$nn	84	2	3
absolute	STY \$nnnn	8C	3	4
zero-page,X	STY \$nn,X	94	2	4

TAS [unintended]

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.

TAS : Munge X Register and Stack Pointer		6502		
$SP \leftarrow A \text{ AND } X, M \leftarrow SP \text{ AND } B3$				
		N Z I C D V E		
			
Addressing Mode	Assembly	Code	Bytes	Cycles
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TAX : Transfer Accumulator Register into the X Register	6502			
X ← A				
	N Z I C D V E + +			
Addressing Mode	Assembly	Code	Bytes	Cycles
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TAY : Transfer Accumulator Register into the Y Register	6502			
Y ← A				
	N Z I C D V E + +			
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	TAY	A8	1	2

TSX

This instruction loads the X Register with the contents of the Stack Pointer High (SPH) Register.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TSX : Transfer Stack Pointer High Register into the X Register					6502
X ← SPH					
			N Z I C D V E		
			+ +		
Addressing Mode	Assembly	Code	Bytes	Cycles	
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TXA : Transfer X Register into the Accumulator Register					6502
A ← X					
			N Z I C D V E		
			+ +		
Addressing Mode	Assembly	Code	Bytes	Cycles	
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					
			N Z I C D V E		
				
Addressing Mode	Assembly	Code	Bytes	Cycles	
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TYA : Transfer Y Register into the Accumulator Register		6502		
A ← Y				
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	TYA	98	1	2

XAA [unintended]

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

XAA : Transfer X into A and AND with operand		6502		
A ← X AND VALUE				
Addressing Mode	Assembly	Code	Bytes	Cycles
immediate 8bit	XAA #\$nn	8B	2	2

4510 INSTRUCTION SET

Instruction Timing

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.

Opcode Table

The coloured cells indicate an extended 45GS02 Opcode. A Q pseudo register opcode is marked blue, a base-page indirect Z indexed opcode that can use 32-bit pointers is cyan.

	\$x0	\$x1	\$x2
\$0x	size OPC mode	cyc QOP Q mode	size FARQ Q lbpZ

The letters attached to the cycle count have the following meaning:

	Meaning
b	Add one cycle if branch is taken. Add one more cycle if branch taken crosses a page boundary.
d	Subtract one cycle when CPU is at 3.5MHz.
i	Add one cycle if clock speed is at 40 MHz.
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.

Opcode Table 4510/45GS02

	\$x0	\$x1	\$x2	\$x3	\$x4	\$x5	\$x6	\$x7
\$0x	1 BRK imp	7 6rp ORA bpX	1 CLE imp	1 SEE imp	2 TSB bp	3r ORA Q bp	2 ASL Q bp	2 4rb RMB0 bp
\$1x	2 BPL rel	2 5rp ORA Q ibpY	2 5rp ORA Q ibpZ	3 BPL relfar	2 TRB bp	5r ORA bpX	2 4r ASL Q bpX	2 4rb RMB1 bp
\$2x	3 JSR abs	5s 2 5rp AND bpX	2 5rp JSR ind	3 5rp JSR indX	2 BIT Q bp	3r 2 3r AND Q bp	2 4r ROL Q bp	2 4r RMB2 bp
\$3x	2 BMI rel	2r 2 5rp AND Q ibpY	2 5rp AND Q ibpZ	3 3b BMI relfar	2 BIT bpX	3rp 2 4rp AND bpX	2 5rp ROL Q bpX	2 4r RMB3 bp
\$4x	1 RTI imp	6m 2 5r EOR bpX	1 NEG acc	1 1 ASR Q acc	2 ASR Q bp	4r 2 3r EOR Q bp	2 4r LSR Q bp	2 4r RMB4 bp
\$5x	2 BVC rel	2 2b 5rp EOR ibpY	2 5rp EOR Q ibpZ	3 3b BVC relfar	2 ASR Q bpX	5rp 2 3rp EOR bpX	2 3rp LSR Q bpX	2 4r RMB5 bp
\$6x	1 RTS imp	6m 2 5r ADC bpX	2 RTS imm	4 3 3b BSR relfar	2 STZ bp	3 2 3r ADC Q bp	2 5r ROR Q bp	2 5r RMB6 bp
\$7x	2 BVS rel	2 2b 5rp ADC ibpY	2 5rp ADC Q ibpZ	3 3b BVS relfar	2 STZ bpX	3 2 3r ADC bpX	2 5rmdp 2 4r ROR Q bpX	2 4r RMB7 bp
\$8x	2 BRA rel	2 2b 5p STA Q bpX	2 6p STA Q ispY	3 3b BRA relfar	2 STY bp	3 2 3r STA Q bp	2 3 2 STX bp	2 4r SMB0 bp
\$9x	2 BCC rel	2 2b 5p STA Q ibpY	2 5p STA Q ibpZ	3 3b BCC relfar	2 STY bpX	3p 2 3p STA Q bpX	2 3p 2 STX bpY	2 4r SMB1 bp
\$Ax	2 LDY imm	2 2 5rp LDA Q bpX	2 LDX imm	2 2 LDZ imm	2 LDY bp	3r 2 3r LDA Q bp	2 3r 2 LDX bp	2 4r SMB2 bp
\$Bx	2 BCS rel	2 2b 5rp LDA ibpY	2 5rp LDA Q ibpZ	3 3b BCS relfar	2 LDY bpX	3rp 2 3rp LDA bpX	2 5rp 2 LDX bpY	2 4r SMB3 bp
\$Cx	2 CPY imm	2 2 5rp CMP Q bpX	2 CPZ imm	2 2 7mdr DEW bp	2 CPY bp	3r 2 3r CMP Q bp	2 5mdr 2 DEC Q bp	2 4r SMB4 bp
\$Dx	2 BNE rel	2 2b 5rp CMP Q ibpY	2 5rp CMP Q ibpZ	3 3b BNE relfar	2 CPZ bp	3r 2 3rp CMP Q bpX	2 5mdrp 2 DEC Q bpX	2 4r SMB5 bp
\$Ex	2 CPX imm	2 2 3pm SBC Q bpX	2 6rmp LDA ispY	2 7mdr INW bp	2 CPX bp	3r 2 3r SBC Q bp	2 5mdr 2 INC Q bp	2 4r SMB6 bp
\$Fx	2 BEQ rel	2 2b 3rp SBC Q ibpY	2 5rp SBC Q ibpZ	3 3b BEQ relfar	3 5m PHW im16	2 3rp 2 3rp SBC Q bpX	2 5dmrp 2 INC Q bpX	2 4r SMB7 bp

Opcode Table 4510/45GS02

\$x8	\$x9	\$xA	\$xB	\$xC	\$xD	\$xE	\$xF	
1 PHP imp	2 ORA imm	2 ASL Q acc	1 TSY imp	1 TSB abs	5r ORA Q abs	3 ASL Q abs	5r BBR0 bpr8	\$0x
1 CLC imp	1 ORA absY	3 INC Q acc	1 INZ imp	1 TRB abs	4r ORA absX	3 ASL Q absX	5rp BBR1 bpr8	\$1x
1 PLP imp	4m AND imm	2 ROL Q acc	1 TYS imp	1 BIT Q abs	4r AND Q abs	3 ROL Q abs	5r BBR2 bpr8	\$2x
1 SEC imp	1 AND absY	3 DEC Q acc	1 DEZ imp	1 BIT absX	4rp AND absX	3 ROL Q absX	4b BBR3 bpr8	\$3x
1 PHA imp	2 EOR imm	2 LSR Q acc	1 TAZ imp	1 JMP abs	3 EOR Q abs	3 LSR Q abs	4rb BBR4 bpr8	\$4x
1 CLI imp	1 EOR absY	3 PHY imp	1 TAB imp	1 MAP imp	3 EOR absX	3 LSR Q absX	4rb BBR5 bpr8	\$5x
1 PLA imp	4m ADC imm	2 ROR Q acc	1 TZA imp	1 JMP ind	5r ADC Q abs	3 ROR Q abs	6r BBR6 bpr8	\$6x
1 SEI imp	1s ADC absY	3 PLY imp	1 TBA imp	1 JMP indx	6mp ADC absX	3 ROR Q absX	5rmdp BBR7 bpr8	\$7x
1 DEY imp	1s BIT imm	2 TXA imp	1 STY absX	3 STY abs	4 STA Q abs	3 STX abs	4br BBS0 bpr8	\$8x
1 TYA imp	1 STA Q absY	3 TXS imp	1 STX absY	3 STZ abs	4 STA Q absX	3 STZ absX	4br BBS1 bpr8	\$9x
1 TAY imp	1 LDA imm	2 TAX imp	1 LDZ abs	3 LDY abs	4r LDA Q abs	3 LDX abs	4br BBS2 bpr8	\$Ax
1 CLV imp	1 LDA absY	3 TSX imp	1 LDZ absX	3 LDY absX	4rp LDA absX	3 LDX absY	4br BBS3 bpr8	\$Bx
1 INY imp	1s CMP imm	2 DEX imp	1s ASW abs	7rmd CPY abs	4r CMP Q abs	3 DEC Q abs	6mdr BBS4 bpr8	\$Cx
1 CLD imp	1 CMP Q absY	3 PHX imp	1 PHZ imp	3 CPZ abs	4r CMP Q absX	3 DEC Q absX	6mdrp BBS5 bpr8	\$Dx
1 INX imp	1s SBC imm	2 EOM imp	1 ROW abs	3 CPX abs	4r SBC Q abs	3 INC Q abs	6dmr BBS6 bpr8	\$Ex
1 SED imp	1s SBC Q absY	3 PLX imp	1 PLZ imp	3 PHW abs	7m SBC Q absX	3 INC Q absX	4br BBS7 bpr8	\$Fx

ADC

This instruction adds the argument and the Carry Flag to the contents of the Accumulator Register. 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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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
		+	+	.	+	.	+	.
Addressing Mode	Assembly	Code	Bytes	Bytes	Cycles			
(indirect),X)	ADC (\$nn,X)	61	2	5	r			
base-page	ADC \$nn	65	2	3	r			
immediate 8bit	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	r			
absolute,Y	ADC \$nnnn,Y	79	3	4	r			
absolute,X	ADC \$nnnn,X	7D	3	4	r			

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

AND : Binary AND		4510	
A ← A AND M		N Z I C D V E	
Addressing Mode	Assembly	Code	Bytes Cycles
(indirect,X)	AND (\$nn,X)	21	2 5 pr
base-page	AND \$nn	25	2 3 r
immediate 8bit	AND #\$nn	29	2 2
absolute	AND \$nnnn	2D	3 4 r
(indirect),Y	AND (\$nn),Y	31	2 5 pr
(indirect),Z	AND (\$nn),Z	32	2 5 pr
base-page,X	AND \$nn,X	35	2 4 pr
absolute,Y	AND \$nnnn,Y	39	3 4 r
absolute,X	AND \$nnnn,X	3D	3 4 pr

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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 Memory or Accumulator**4510** $A \leftarrow A \lll 1$ or $M \leftarrow M \lll 1$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

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

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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 $M \leftarrow M >> 1$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
accumulator	ASR A	43	1	1	<i>s</i>
base-page	ASR \$nn	44	2	4	<i>r</i>
base-page,X	ASR \$nn,X	54	2	5	<i>pr</i>

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.

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The C flag will be set if bit 7 of the upper byte was set, prior to being shifted, otherwise it will be cleared.

ASW : Arithmetic Shift Word Left		4510
$M \leftarrow M \ll 1$		
		N Z I C D V E + + . + . . .
Addressing Mode	Assembly	Code
absolute	ASW \$nnnn	CB

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.

BBRO

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

BBR0 : Branch on Bit 0 Reset	4510
$M(0)=0 \implies PC \leftarrow PC + R8$	
	N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page+rel	BBR0 \$nn,\$rr	0F	3	0 ^b r

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.

BBR1 : Branch on Bit 1 Reset	4510
$M(1)=0 \implies PC \leftarrow PC + R8$	
	N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page+rel	BBR1 \$nn,\$rr	1F	3	5 ^b

b Add one cycle if branch is taken.

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

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
$M(2)=0 \implies PC \leftarrow PC + R8$	
	N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page+rel	BBR2 \$nn,\$rr	2F	3	5 ^b

b Add one cycle if branch is taken.

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

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		
$M(3)=0 \implies PC \leftarrow PC + R8$		N Z I C D V E		
Addressing Mode	Assembly	Code	Bytes	Cycles
base-page+rel	BBR3 \$nn,\$rr	3F	3	4 ^b

b Add one cycle if branch is taken.

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

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		
$M(4)=0 \implies PC \leftarrow PC + R8$		N Z I C D V E		
Addressing Mode	Assembly	Code	Bytes	Cycles
base-page+rel	BBR4 \$nn,\$rr	4F	3	4 ^{br}

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.

BBR5 : Branch on Bit 5 Reset	4510				
$M(5)=0 \implies PC \leftarrow PC + R8$					
	N Z I C D V E				
Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page+rel	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.

BBR6 : Branch on Bit 6 Reset	4510				
$M(6)=0 \implies PC \leftarrow PC + R8$					
	N Z I C D V E				
Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page+rel	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
$M(7)=0 \implies PC \leftarrow PC + R8$	
	N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page+rel	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
$M(0)=1 \implies PC \leftarrow PC + R8$	
	N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page+rel	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
$M(1)=1 \implies PC \leftarrow PC + R8$		
	N Z I C D V E	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page+rel	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.

BBS2 : Branch on Bit 2 Set		4510
$M(2)=1 \implies PC \leftarrow PC + R8$		
	N Z I C D V E	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page+rel	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.

BBS3 : Branch on Bit 3 Set	4510			
$M(3)=1 \implies PC \leftarrow PC + R8$				
	N Z I C D V E			
Addressing Mode	Assembly	Code	Bytes	Cycles
base-page+rel	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			
$M(4)=1 \implies PC \leftarrow PC + R8$				
	N Z I C D V E			
Addressing Mode	Assembly	Code	Bytes	Cycles
base-page+rel	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
$M(5)=1 \implies PC \leftarrow PC + R8$		
	N Z I C D V E	

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page+rel	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
$M(6)=1 \implies PC \leftarrow PC + R8$		
	N Z I C D V E	

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page+rel	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.

BBS7 : Branch on Bit 7 Set

4510

 $M(7)=1 \implies PC \leftarrow PC + R8$

N Z I C D V E

.

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page+rel	BBS7 \$nn,\$rr	FF	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.**BCC**

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

BCC : Branch on Carry Flag Clear

4510

 $C=0 \implies PC \leftarrow PC + R8 \text{ or } PC \leftarrow PC + R16$

N Z I C D V E

.

Addressing Mode	Assembly	Code	Bytes	Cycles	
relative	BCC \$rr	90	2	2	<i>b</i>
16-bit relative	BCC \$rrrr	93	3	3	<i>b</i>

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

 $C=1 \implies PC \leftarrow PC + R8 \text{ or } PC \leftarrow PC + R16$

N Z I C D V E

.

Addressing Mode	Assembly	Code	Bytes	Cycles	
relative	BCS \$rr	B0	2	2	<i>b</i>
16-bit relative	BCS \$rrrr	B3	3	3	<i>b</i>

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. BEW stands for branch if equal, because a CMP will result in the zero flag being set if the operants are equal.

BEQ : Branch on Zero Flag Set		4510			
Z=1 \Rightarrow PC \leftarrow PC + R8 or PC \leftarrow PC + R16		N Z I C D V E			
Addressing Mode	Assembly	Code	Bytes	Cycles	
relative	BEQ \$rr	F0	2	2	^b
16-bit relative	BEQ \$rrrr	F3	3	3	^b

^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, otherwise it will be cleared.
- The V flag will be set if the bit 6 of the memory location is set, otherwise it will be cleared.
- The Z flag will be set if the result of A AND M is zero, otherwise it will be cleared.

BIT : Perform Bit Test

4510

 $N \leftarrow M(7), V \leftarrow M(6), Z \leftarrow A \text{ AND } M$

N	Z	I	C	D	V	E
+	+	.	.	.	+	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
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 8bit	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. BMI stands for branch on minus.

BMI : Branch on Negative Flag Set

4510

 $N=1 \implies PC \leftarrow PC + R8 \text{ or } PC \leftarrow PC + R16$

N	Z	I	C	D	V	E
.

Addressing Mode	Assembly	Code	Bytes	Cycles	
relative	BMI \$rr	30	2	2	<i>r</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.

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

This instruction branches to the indicated address if the Zero Flag is clear. BNE stands for Branch if not equal, because a CMP will result in the zero flag being cleared if the operands are not equal.

BNE : Branch on Zero Flag Clear**4510** $Z=0 \implies PC \leftarrow PC + R8 \text{ or } PC \leftarrow PC + R16$ **N Z I C D V E**

Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BNE \$rr	D0	2	2 ^b
16-bit relative	BNE \$rrrr	D3	3	3 ^b

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 stands for branch on plus.

BPL : Branch on Negative Flag Clear**4510** $N=0 \implies PC \leftarrow PC + R8 \text{ or } PC \leftarrow PC + R16$ **N Z I C D V E**

Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BPL \$rr	10	2	2 ^b
16-bit relative	BPL \$rrrr	13	3	3 ^b

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

Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BRA \$rr	80	2	2 ^b
16-bit relative	BRA \$rrrr	83	3	3 ^b

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

STACK \leftarrow PC + 2; PC \leftarrow (\$FFFF)

N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	BRK	00	1	7

BSR

This instruction branches to the indicated address, saving the address of the following instruction 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
STACK \leftarrow PC + len(V), PC \leftarrow PC + V		N Z I C D V E
	
Addressing Mode	Assembly	Code
16-bit relative	BSR \$rrrr	63

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
V=0 \implies PC \leftarrow PC + R8 or PC \leftarrow PC + R16		N Z I C D V E
	
Addressing Mode	Assembly	Code
relative	BVC \$rr	50
16-bit relative	BVC \$rrrr	53

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
 $V=1 \implies PC \leftarrow PC + R8$ or $PC \leftarrow PC + R16$

4510

N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
relative	BVS \$rr	70	2	2 ^b
16-bit relative	BVS \$rrrr	73	3	3 ^b

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

$C \leftarrow 0$

N Z I C D V E
 . . . + . . .

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	CLC	18	1	1 ^s

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	Cycles
implied	CLD	D8	1	1 <i>s</i>

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

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.

CLE : Clear Extended Stack Disable Flag	4510			
E ← 0				
	N Z I C D V E · · · + · ·			
Addressing Mode	Assembly	Code	Bytes	Cycles
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.

CLI : Clear Interrupt Disable Flag	4510			
$I \leftarrow 0$				
	N Z I C D V E + +			
Addressing Mode	Assembly	Code	Bytes	Cycles
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.

CLV : Clear Overflow Flag	4510			
$V \leftarrow 0$				
	N Z I C D V E +			
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	CLV	B8	1	1 <i>s</i>

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

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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The Z flag will be set if the result of $A - M$ is zero, otherwise it will be cleared.

CMP : Compare Accumulator

4510

 $N, C, Z \Leftarrow [A - M]$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
(indirect,X)	CMP (\$nn,X)	C1	2	5	<i>pr</i>
base-page	CMP \$nn	C5	2	3	<i>r</i>
immediate 8bit	CMP #\$nn	C9	2	2	
absolute	CMP \$nnnn	CD	3	4	<i>r</i>
(indirect),Y	CMP (\$nn),Y	D1	2	5	<i>pr</i>
(indirect),Z	CMP (\$nn),Z	D2	2	5	<i>pr</i>
base-page,X	CMP \$nn,X	D5	2	3	<i>pr</i>
absolute,Y	CMP \$nnnn,Y	D9	3	4	<i>pr</i>
absolute,X	CMP \$nnnn,X	DD	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.

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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The Z flag will be set if the result of $X - M$ is zero, otherwise it will be cleared.

CPX : Compare X Register

4510

 $N,C,Z \Leftarrow [X - M]$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles
immediate 8bit	CPX #\$nn	E0	2	2
base-page	CPX \$nn	E4	2	3 <i>r</i>
absolute	CPX \$nnnn	EC	3	4 <i>r</i>

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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The Z flag will be set if the result of $Y - M$ is zero, otherwise it will be cleared.

CPY : Compare Y Register

4510

 $N,C,Z \Leftarrow [Y - M]$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles
immediate 8bit	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. has its most significant bit set, otherwise 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 , otherwise it will be cleared.
- The Z flag will be set if the result of $Z - M$ is zero, otherwise it will be cleared.

CPZ : Compare Z Register		4510
$N, C, Z \Leftarrow [Z - M]$		
		N Z I C D V E
		+ + . + . . .
Addressing Mode	Assembly	Code
immediate 8bit	CPZ #\$nn	C2
base-page	CPZ \$nn	D4
absolute	CPZ \$nnnn	DC

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

DEC : Decrement

4510

 $A \leftarrow A - 1$ or $M \leftarrow M - 1$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
accumulator	DEC A	3A	1	1	<i>s</i>
base-page	DEC \$nn	C6	2	5	<i>dmr</i>
absolute	DEC \$nnnn	CE	3	6	<i>dmr</i>
base-page,X	DEC \$nn,X	D6	2	5	<i>dmpr</i>
absolute,X	DEC \$nnnn,X	DE	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.**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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

DEW : Decrement Memory Word

4510

 $M16 \leftarrow M16 - 1$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

DEX : Decrement X Register		4510		
$X \leftarrow X - 1$				
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	DEX	CA	1	1 ^s

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

DEY : Decrement Y Register		4510		
$Y \leftarrow Y - 1$				
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	DEY	88	1	1 ^s

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

DEZ : Decrement Z Register	4510			
Z \leftarrow Z - 1				
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	DEZ	3B	1	1 ^s

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 addressing mode causes a two cycle penalty, caused by the time required to read the extra two bytes of the pointer.

NOTE: please take care if you use EOM/NOP after a Hypervisor Call for delay, as this might change your next instruction. CLV can be used as an alternative.

Side effects

- Removes the prohibition on all interrupts caused by the the MAP instruction, allowing Non-Maskable Interrupts to again occur, and IRQ interrupts, if the Interrupt Disable Flag is not set.

EOM : End of Mapping Sequence / No-Operation

4510

Special

N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	EOM	EA	1	1 ^s

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

EOR

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, but not both.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

EOR : Binary Exclusive OR

4510

A ← A XOR M

N Z I C D V E

+ +

Addressing Mode	Assembly	Code	Bytes	Cycles
(indirect,X)	EOR (\$nn,X)	41	2	5 ^r
base-page	EOR \$nn	45	2	3 ^r
immediate 8bit	EOR #\$nn	49	2	2
absolute	EOR \$nnnn	4D	3	4 ^r
(indirect),Y	EOR (\$nn),Y	51	2	5 ^{pr}
(indirect),Z	EOR (\$nn),Z	52	2	5 ^{pr}
base-page,X	EOR \$nn,X	55	2	3 ^p
absolute,Y	EOR \$nnnn,Y	59	3	4 ^{pr}
absolute,X	EOR \$nnnn,X	5D	3	4 ^{pr}

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

INC : Increment Memory or Accumulator		4510				
$A \leftarrow A + 1$ or $M \leftarrow M + 1$		NZICDVE				
Addressing Mode	Assembly	Code	Bytes	Cycles		
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 IEW \$12 would cause memory location to be set to \$79.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

INW : Increment Memory Word**4510** $M16 \leftarrow M16 + 1$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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	<i>s</i>

s 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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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 <i>s</i>

s 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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

INZ : Increment Z Register	4510			
$Z \leftarrow Y + 1$				
	N Z I C D V E + +			
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	INZ	1B	1	1 <i>s</i>

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

JMP

This instruction sets the Program Counter (PC) Register to the address indicated by the instruction, causing execution to continue from that address.

JMP : Jump to Address

4510

PC ← M2:M1

N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
absolute	JMP \$nnnn	4C	3	3
(indirect)	JMP (\$nnnn)	6C	3	5 <i>r</i>
(indirect,X)	JMP (\$nnnn,X)	7C	3	6 <i>mp</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.**JSR**

This instruction saves the address of the instruction following the JSR instruction onto the stack, and then sets the Program 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 Program Counter (PC) register.

JSR : Jump to Sub-Routine

4510

PC ← M2:M1, STACK ← PCH:PCL

N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles
absolute	JSR \$nnnn	20	3	5
(indirect)	JSR (\$nnnn)	22	3	5 <i>r</i>
(indirect,X)	JSR (\$nnnn,X)	23	3	5 <i>pr</i>

p Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

LDA : Load Accumulator			4510
A ← M		N Z I C D V E	
Addressing Mode	Assembly	Code	Bytes Cycles
(indirect,X)	LDA (\$nn,X)	A1	2 5 <i>pr</i>
base-page	LDA \$nn	A5	2 3 <i>r</i>
immediate 8bit	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>
(immediate,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, i.e. has its most significant bit set, otherwise it will be cleared.

- The Z flag will be set if the result is zero, otherwise 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 8bit	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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

LDY : Load Y Register		4510				
Y ← M		N Z I C D V E				
		+ +				
Addressing Mode	Assembly	Code	Bytes	Cycles		
immediate 8bit	LDY #\$nn	A0	2	2		
base-page	LDY \$nn	A4	2	3	r	
absolute	LDY \$nnnn	AC	3	4	r	
base-page,X	LDY \$nn,X	B4	2	3	pr	
absolute,X	LDY \$nnnn,X	BC	3	4	pr	

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

LDZ : Load Z Register		4510					
$Z \leftarrow M$							
		N Z I C D V E					
Addressing Mode	Assembly	Code	Bytes	Bytes	Cycles		
immediate 8bit	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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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** $A \leftarrow A \gg 1, C \leftarrow A(0)$ or $M \leftarrow M \gg 1, C \leftarrow M(0)$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page	LSR \$nn	46	2	4	<i>r</i>
accumulator	LSR A	4A	1	1	<i>s</i>
absolute	LSR \$nnnn	4E	3	5	<i>r</i>
base-page,X	LSR \$nn,X	56	2	3	<i>pr</i>
absolute,X	LSR \$nnnn,X	5E	3	5	<i>pr</i>

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**

Special

N	Z	I	C	D	V	E
.

Addressing Mode	Assembly	Code	Bytes	Cycles	
implied	MAP	5C	1	1	<i>s</i>

s 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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

NEG : Negate Accumulator	4510			
A $\leftarrow (A \text{ XOR } \$FF) + 1$				
Addressing Mode	Assembly	Code	Bytes	Cycles
accumulator	NEG A	42	1	1 <i>s</i>

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

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

ORA : Binary OR**4510**A \leftarrow A OR M

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
(indirect,X)	ORA (\$nn,X)	01	2	6	<i>pr</i>
base-page	ORA \$nn	05	2	3	<i>r</i>
immediate 8bit	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>r</i>
absolute,Y	ORA \$nnnn,Y	19	3	4	<i>r</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

N	Z	I	C	D	V	E
.

Addressing Mode	Assembly	Code	Bytes	Cycles	
implied	PHA	48	1	2	

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	2

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 16bit	PHW #\$nnnn	F4	3	5 <i>m</i>
absolute	PHW \$nnnn	FC	3	7 <i>m</i>

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 <i>m</i>

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.

PHY : Push Y Register onto the Stack		4510
STACK \leftarrow Y, SP \leftarrow SP - 1		N Z I C D V E
	
Addressing Mode	Assembly	Code
implied	PHY	5A
Bytes	Cycles	
1	2	

PHZ

This instruction pushes the contents of the Z Register onto the stack, and decrements the value of the Stack Pointer by 1.

PHZ : Push Z Register onto the Stack		4510
STACK \leftarrow z, SP \leftarrow SP - 1		N Z I C D V E
	
Addressing Mode	Assembly	Code
implied	PHZ	DB
Bytes	Cycles	
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.

PLA : Pull Accumulator Register from the Stack		4510
A \leftarrow STACK, SP \leftarrow SP + 1		N Z I C D V E
		++.....
Addressing Mode	Assembly	Code
implied	PLA	68
Bytes	Cycles	
1	4	<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)

PLP : Pull Processor Flags from the Stack		4510		
A ← STACK, SP ← SP + 1				
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	PLP	28	1	4 ^m

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.

PLX : Pull X Register from the Stack		4510		
X ← STACK, SP ← SP + 1				
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	PLX	FA	1	4 ^m

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** $Y \leftarrow \text{STACK}, SP \leftarrow SP + 1$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	PLY	7A	1	4 ^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** $Z \leftarrow \text{STACK}, SP \leftarrow SP + 1$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	PLZ	FB	1	4 ^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** $M(0) \leftarrow 0$

N	Z	I	C	D	V	E
.

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page	RMBO \$nn	07	2	4 ^{br}

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.

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
base-page	RMB1 \$nn	17
		2 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.

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
base-page	RMB2 \$nn	27
		2 4 <i>r</i>

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 <i>r</i>

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 <i>r</i>

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 <i>r</i>

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
base-page	RMB6 \$nn	67
		2 5 <i>r</i>

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
base-page	RMB7 \$nn	77
		2 4 <i>r</i>

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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

 $M \leftarrow M \ll 1, C \leftarrow M(7), M(0) \leftarrow C$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page	ROL \$nn	26	2	4	<i>r</i>
accumulator	ROL A	2A	1	1	<i>s</i>
absolute	ROL \$nnnn	2E	3	5	<i>r</i>
base-page,X	ROL \$nn,X	36	2	5	<i>pr</i>
absolute,X	ROL \$nnnn,X	3E	3	5	<i>pr</i>

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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise 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** $M \leftarrow M \gg 1, C \leftarrow M(0), M(7) \leftarrow C$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page	ROR \$nn	66	2	5	<i>r</i>
accumulator	ROR A	6A	1	1	<i>s</i>
absolute	ROR \$nnnn	6E	3	6	<i>r</i>
base-page,X	ROR \$nn,X	76	2	5	<i>dmp</i>
absolute,X	ROR \$nnnn,X	7E	3	5	<i>dmp</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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The C flag will be set if bit 7 of the upper byte was set, prior to being shifted.

ROW : Rotate Word Left**4510** $M2:M1 \leftarrow M2:M1 \ll 1, C \leftarrow M2(7), M1(0) \leftarrow C$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
absolute	ROW \$nnnn	EB	3	5	<i>dmp</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 Program Counter (PC) register from the stack, allowing an interrupted program 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.

RTI : Return From Interrupt		4510
P ← STACK, PC ← STACK		N Z I C D V E
		+ + + + + +
Addressing Mode	Assembly	Code
implied	RTI	40
		Bytes Cycles
		1 6 m

m Subtract non-bus cycles when at 40MHz.

RTS

This instruction adds an optional argument to the Stack Pointer (SP) Register, and then pops the Program Counter (PC) register from the stack, allowing a routine to return to its caller.

RTS : Return From Subroutine		4510
PC ← STACK or PC ← STACK + M, SP ← SP – 2		N Z I C D V E
	
Addressing Mode	Assembly	Code
implied	RTS	60
immediate 8bit	RTS #\$nn	62
		Bytes Cycles
		1 6 m
		2 4

m Subtract non-bus cycles when at 40MHz.

SBC

This instruction performs A – M – 1 + C, and sets the processor flags accordingly. The result is stored in the Accumulator Register.

NOTE: 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 of $A - M$ is negative, i.e. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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, otherwise it will be cleared.

SBC : Subtract With Carry			4510		
Addressing Mode	Assembly	Code	Bytes	Cycles	
(indirect,X)	SBC (\$nn,X)	E1	2	3	<i>mp</i>
base-page	SBC \$nn	E5	2	3	<i>r</i>
immediate 8bit	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	5	<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 ← 1				
	N Z I C D V E . . . + . . .			
Addressing Mode	Assembly	Code	Bytes	Cycles
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 <i>s</i>

s 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** $E \leftarrow 1$ **N Z I C D V E**

. . . . +

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	SEE	03	1	1 ^s

s 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.

SEI : Set Interrupt Disable Flag**4510** $I \leftarrow 1$ **N Z I C D V E**

. . + . . .

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	SEI	78	1	1 ^s

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

SMBO

This instruction sets bit zero of the indicated address. No flags are modified, regardless of the result.

SMBO : Set Bit 0 in Base Page**4510** $M(0) \leftarrow 1$ **N Z I C D V E**

.

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page	SMBO \$nn	87	2	4 ^r

r 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.

SMB1 : Set Bit 1 in Base Page		4510
$M(1) \leftarrow 1$		
N Z I C D V E		
.		
Addressing Mode	Assembly	Code
base-page	SMB1 \$nn	97
Bytes	Cycles	
2	4	<i>r</i>

r 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.

SMB2 : Set Bit 2 in Base Page		4510
$M(2) \leftarrow 1$		
N Z I C D V E		
.		
Addressing Mode	Assembly	Code
base-page	SMB2 \$nn	A7
Bytes	Cycles	
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.

SMB3 : Set Bit 3 in Base Page		4510
$M(3) \leftarrow 1$		
N Z I C D V E		
.		
Addressing Mode	Assembly	Code
base-page	SMB3 \$nn	B7
Bytes	Cycles	
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.

SMB4 : Set Bit 4 in Base Page		4510
$M(4) \leftarrow 1$		
N Z I C D V E		
.		
Addressing Mode	Assembly	Code
base-page	SMB4 \$nn	C7
	Bytes	Cycles
	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.

SMB5 : Set Bit 5 in Base Page		4510
$M(5) \leftarrow 1$		
N Z I C D V E		
.		
Addressing Mode	Assembly	Code
base-page	SMB5 \$nn	D7
	Bytes	Cycles
	2	4 <i>r</i>

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

SMB6

This instruction sets bit 6 of the indicated address. No flags are modified, regardless of the result.

SMB6 : Set Bit 6 in Base Page		4510
$M(6) \leftarrow 1$		
N Z I C D V E		
.		
Addressing Mode	Assembly	Code
base-page	SMB6 \$nn	E7
	Bytes	Cycles
	2	4 <i>r</i>

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

SMB7

This instruction sets bit 7 of the indicated address. No flags are modified, regardless of the result.

SMB7 : Set Bit 7 in Base Page		4510
M(7) ← 1		
N Z I C D V E		
.		
Addressing Mode	Assembly	Code
base-page	SMB7 \$nn	F7

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

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
(indirect,X)	STA (\$nn,X)	81
(immediate,SP),Y	STA (\$nn,SP),Y	82
base-page	STA \$nn	85
absolute	STA \$nnnn	8D
(indirect),Y	STA (\$nn),Y	91
(indirect),Z	STA (\$nn),Z	92
base-page,X	STA \$nn,X	95
absolute,Y	STA \$nnnn,Y	99
absolute,X	STA \$nnnn,X	9D

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

 $M \leftarrow X$ **N Z I C D V E**

.

Addressing Mode	Assembly	Code	Bytes	Cycles
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

 $M \leftarrow Y$ **N Z I C D V E**

.

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page	STY \$nn	84	2	3
absolute,X	STY \$nnnn,X	8B	3	4 <i>p</i>
absolute	STY \$nnnn	8C	3	4
base-page,X	STY \$nn,X	94	2	3 <i>p</i>

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	
$M \leftarrow Z$		N Z I C D V E	
Addressing Mode	Assembly	Code	Bytes Cycles
base-page	STZ \$nn	64	2 3
base-page,X	STZ \$nn,X	74	2 3
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	
Addressing Mode	Assembly	Code	Bytes Cycles
implied	TAB	5B	1 1 ^s

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TAX : Transfer Accumulator Register into the X Register 4510

X ← A

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	TAX	AA	1	1 ^s

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TAY : Transfer Accumulator Register into the Y Register 4510

Y ← A

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	TAY	A8	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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TAZ : Transfer Accumulator Register into the Z Register 4510

Z ← A

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	TAZ	4B	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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TBA : Transfer Base Page Register into the Accumulator 4510

A ← B

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles
implied	TBA	7B	1	1 ^s

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
$M \leftarrow M \text{ AND } (\text{NOT } A)$		
		N Z I C D V E
		. +
Addressing Mode	Assembly	Code
base-page	TRB \$nn	14
absolute	TRB \$nnnn	1C

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
		. +
Addressing Mode	Assembly	Code
base-page	TSB \$nn	04
absolute	TSB \$nnnn	0C

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 (SPH) Register.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TSX : Transfer Stack Pointer High Register into the X Register	4510
---	-------------

X ← SPH

N Z I C D V E
+ +

Addressing Mode	Assembly	Code	Bytes	Cycles
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TSY : Transfer Stack Pointer High Register into the Y Register	4510
---	-------------

Y ← SPH

N Z I C D V E
+ +

Addressing Mode	Assembly	Code	Bytes	Cycles
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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TXA : Transfer X Register into the Accumulator Register		4510		
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		
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, i.e. has its most significant bit set, otherwise it will be cleared.

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

TYA : Transfer Y Register into the Accumulator Register	4510			
A ← Y				
	N Z I C D V E + +			
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	TYA	98	1	1 <small>s</small>

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.

TYS : Transfer Y Register into Stack Pointer High Register	4510			
SPH ← Y				
	N Z I C D V E			
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	TYS	2B	1	1 <small>s</small>

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

TZA : Transfer Z Register into the Accumulator Register	4510			
A ← Z				
	N Z I C D V E + + . . .			
Addressing Mode	Assembly	Code	Bytes	Cycles
implied	TZA	6B	1	1 ^s

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

45GS02 COMPOUND INSTRUCTIONS

As the 4510 has no unallocated opcodes, the 45GS02 uses compound instructions to implement its extension. These compound instructions consist of one or more single byte instructions placed immediately before a conventional instruction. These prefixes instruct the 45GS02 to treat the following instruction differently, as described in Chapter/Appendix [G on page G-3](#).

You can find them highlighted in the 4510 Opcode Table (see Chapter/Appendix [H on page H-62](#))

ADC

This instruction adds the argument and the Carry Flag to the contents of the Accumulator Register. 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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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

45GS02

$$A \leftarrow A + M + C$$

N	Z	I	C	D	V	E
+	+	.	+	.	+	.

Addressing Mode	Assembly	Code	Bytes	Cycles
[indirect],Z	ADC [\$nn],Z	EA 72	3	7 <i>ipr</i>

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

p Add one cycle if indexing crosses a page boundary.

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

ADCQ

This instruction adds the argument and the Carry Flag to the contents of the 32-bit Q Pseudo Register.

NOTE: the indicated memory location is treated as the first byte of a 32-bit little-endian value.

NOTE: If the D flag is set, the operation is undefined and subject to change.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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 $\geq 2^{32}$.

ADCQ : Add with carry Quad

45GS02

 $A \leftarrow A + M + C$

N	Z	I	C	D	V	E
+	+	.	+	.	+	.

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page quad	ADCQ \$nn	42 42 65	4	8 <i>r</i>
absolute quad	ADCQ \$nnnn	42 42 6D	5	9 <i>r</i>
(indirect quad)	ADCQ (\$nn)	42 42 72	4	10 <i>ipr</i>
[indirect quad]	ADCQ [\$nn]	42 42 EA 72	5	13 <i>ipr</i>

i Add one cycle if clock speed is at 40 MHz.*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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

AND : Binary AND

45GS02

 $A \leftarrow A \text{ AND } M$

N	Z	I	C	D	V	E
+	+	.	•	•	•	.

Addressing Mode	Assembly	Code	Bytes	Cycles
[indirect],Z	AND [\$nn],Z	EA 32	3	7 <i>ipr</i>

i Add one cycle if clock speed is at 40 MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

ANDQ

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

NOTE: the indicated memory location is treated as the first byte of a 32-bit little-endian value.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

ANDQ : Binary AND Quad		45GS02					
$Q \leftarrow Q \text{ AND } M$							
N Z I C D V E							
		+	+
Addressing Mode	Assembly	Code	Bytes	Cycles			
base-page quad	ANDQ \$nn	42 42 25	4	8	<i>r</i>		
absolute quad	ANDQ \$nnnn	42 42 2D	5	9	<i>r</i>		
(indirect quad)	ANDQ (\$nn)	42 42 32	4	10	<i>i pr</i>		
[indirect quad]	ANDQ [\$nn]	42 42 EA 32	5	13	<i>i pr</i>		

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

p Add one cycle if indexing crosses a page boundary.

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

ASLQ

This instruction shifts either the Q pseudo-register or contents of the provided memory location and following three one bit left, treating them as holding a little-endian 32-bit value. Bit 0 will be set to zero, and the bit 31 will be shifted out into the Carry Flag

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.

- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The C flag will be set if bit 31 of the value was set, prior to being shifted, otherwise it will be cleared.

ASLQ : Arithmetic Shift Left Quad		45GS02					
		$Q \leftarrow Q \ll 1$ or $M \leftarrow M \ll 1$					
		N Z I C D V E + + . + . . .					
Addressing Mode	Assembly	Code	Bytes	Cycles			
base-page quad	ASLQ \$nn	42 42 06	4	12	<i>dmr</i>		
Q Pseudo Register	ASLQ Q	42 42 0A	3	3			
absolute quad	ASLQ \$nnnn	42 42 0E	5	13	<i>dmr</i>		
base-page quad,X	ASLQ \$nn,X	42 42 16	4	12	<i>dmpr</i>		
absolute quad,X	ASLQ \$nnnn,X	42 42 1E	5	13	<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.

ASRQ

This instruction shifts either the Q pseudo-register or contents of the provided memory location and following three one bit right, treating them as holding a little-endian 32-bit value. Bit 31 is considered to be a sign bit, and is preserved. The content 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. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The C flag will be set if bit 0 of the value was set, prior to being shifted, otherwise it will be cleared.

ASRQ : Arithmetic Shift Right Quad

45GS02

 $Q \leftarrow Q >> 1$ or $M \leftarrow M >> 1$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles
Q Pseudo Register	ASRQ Q	42 42 43	3	3
base-page quad	ASRQ \$nn	42 42 44	4	12 <i>dmr</i>
base-page quad,X	ASRQ \$nn,X	42 42 54	4	12 <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.**BITQ**

This instruction is used to test the bits stored in a memory location and following three, treating them as holding a little-endian 32-bit value. Bits 30 and 31 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 Q Register and the contents of the indicated memory location.

Side effects

- The N flag will be set if the bit 31 of the memory location is set, otherwise it will be cleared.
- The V flag will be set if the bit 30 of the memory location is set, otherwise it will be cleared.
- The Z flag will be set if the result of Q AND M is zero, otherwise it will be cleared.

BITQ : Perform Bit Test Quad

45GS02

 $N \leftarrow M(31)$, $V \leftarrow M(30)$, $Z \leftarrow Q \text{ AND } M$

N	Z	I	C	D	V	E
+	+	.	•	•	+	.

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page quad	BITQ \$nn	42 42 24	4	8 <i>r</i>
absolute quad	BITQ \$nnnn	42 42 2C	5	9 <i>r</i>

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

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. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The Z flag will be set if the result of $A - M$ is zero, otherwise it will be cleared.

CMP : Compare Accumulator		45GS02
$N,C,Z \Leftarrow [A - M]$		
		N Z I C D V E
		+ + . + . . .
Addressing Mode	Assembly	Code
[indirect],Z	CMP [\$nn],Z	EA D2
		3 7 <i>ipr</i>

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

p Add one cycle if indexing crosses a page boundary.

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

CMPQ

This instruction performs $Q - M$, and sets the processor flags accordingly, but does not modify the contents of the Q Register.

NOTE: the indicated memory location is treated as the first byte of a 32-bit little-endian value.

Side effects

- The N flag will be set if the result of $A - M$ is negative, i.e. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The Z flag will be set if the result of $A - M$ is zero, otherwise it will be cleared.

CMPQ : Compare Q Pseudo Register

45GS02

 $N, C, Z \Leftarrow [Q - M]$

N	Z	I	C	D	V	E
+	+	.	+	.	.	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page quad	CMPQ \$nn	42 42 C5	4	8	<i>r</i>
absolute quad	CMPQ \$nnnn	42 42 CD	5	9	<i>r</i>
(indirect quad)	CMPQ (\$nn)	42 42 D2	4	10	<i>ipr</i>
[indirect quad]	CMPQ [\$nn]	42 42 EA D2	5	13	<i>ipr</i>

i Add one cycle if clock speed is at 40 MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

DEQ

This instruction decrements the Q psuedo register or indicated memory location.

NOTE: the indicated memory location is treated as the first byte of a 32-bit little-endian value.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

DEQ : Decrement Quad

45GS02

 $Q \leftarrow Q - 1$ or $M \leftarrow M - 1$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
Q Pseudo Register	DEQ Q	42 42 3A	3	3	
base-page quad	DEQ \$nn	42 42 C6	4	12	<i>dmr</i>
absolute quad	DEQ \$nnnn	42 42 CE	5	13	<i>dmr</i>
base-page quad,X	DEQ \$nn,X	42 42 D6	4	12	<i>dmpr</i>
absolute quad,X	DEQ \$nnnn,X	42 42 DE	5	13	<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.

FOR

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, but not both.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

EOR : Binary Exclusive OR

45GS02

 $A \leftarrow A \text{ XOR } M$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
[indirect],Z	EOR [\$nn],Z	EA 52	3	7	<i>ipr</i>

i Add one cycle if clock speed is at 40 MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

EORQ

This instruction performs a binary exclusive OR operation of the argument with the Q pseudo register, and stores the result in the Q pseudo register. Only bits that were already set in the Q pseudo register, or that are set in the argument will be set in the accumulator on completion, but not bits that were set in both.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

EORQ : Binary Exclusive OR Quad		45GS02					
$Q \leftarrow Q \text{ XORM } M$							
		N Z I C D V E					
Addressing Mode	Assembly	Code	Bytes	Cycles			
base-page quad	EORQ \$nn	42 42 45	4	8	<i>r</i>		
absolute quad	EORQ \$nnnn	42 42 4D	5	9	<i>r</i>		
(indirect quad)	EORQ (\$nn)	42 42 52	4	10	<i>ipr</i>		
[indirect quad]	EORQ [\$nn]	42 42 EA 52	5	13	<i>ipr</i>		

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

p Add one cycle if indexing crosses a page boundary.

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

INQ

This instruction increments the Q pseudo register or indicated memory location.

Note that the indicated memory location is treated as the first byte of a 32-bit little-endian value.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

INQ : Increment Memory or Accumulator

45GS02

 $Q \leftarrow Q + 1$ or $M \leftarrow M + 1$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
Q Pseudo Register	INQ Q	42 42 1A	3	3	
base-page quad	INQ \$nn	42 42 E6	4	13	<i>dmr</i>
absolute quad	INQ \$nnnn	42 42 EE	5	14	<i>dmr</i>
base-page quad,X	INQ \$nn,X	42 42 F6	4	13	<i>dmpr</i>
absolute quad,X	INQ \$nnnn,X	42 42 FE	5	14	<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.

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

LDA : Load Accumulator

45GS02

 $A \leftarrow M$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles	
[indirect],Z	LDA [\$nn],Z	EA B2	3	7	<i>ipr</i>

i Add one cycle if clock speed is at 40 MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

LDQ

This instruction loads the Q pseudo register with the indicated value, or with the contents of the indicated location. As the Q register is an alias for A, X, Y and Z used together, this operation will set those four registers. A contains the least significant bits, X the next least significant, then Y, and Z contains the most significant bits.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

LDQ : Load Q Pseudo Register		45GS02						
$Q \leftarrow M$		N	Z	I	C	D	V	E
Addressing Mode	Assembly	Code	Bytes	Cycles				
base-page quad	LDQ \$nn	42 42 A5	4	8	r			
absolute quad	LDQ \$nnnn	42 42 AD	5	9	r			
(indirect quad),Z	LDQ (\$nn),Z	42 42 B2	4	10	ipr			
[indirect quad],Z	LDQ [\$nn],Z	42 42 EA B2	5	13	ipr			

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

p Add one cycle if indexing crosses a page boundary.

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

LSRQ

This instruction shifts either the Q pseudo register or contents of the provided memory location one bit right. Bit 31 will be set to zero, and the bit 0 will be shifted out into the Carry Flag.

Note that the memory address is treated as the first address of a little-endian encoded 32-bit value.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

- The C flag will be set if bit 0 of the value was set, prior to being shifted, otherwise it will be cleared.

LSRQ : Logical Shift Right Quad		45GS02					
$Q \leftarrow Q \gg 1, C \leftarrow A(0)$ or $M \leftarrow M \gg 1$		N Z I C D V E					
		$+ + \cdot + \cdot \cdot$					
Addressing Mode	Assembly	Code	Bytes	Cycles			
base-page quad	LSRQ \$nn	42 42 46	4	12	<i>dmr</i>		
Q Pseudo Register	LSRQ Q	42 42 4A	3	3			
absolute quad	LSRQ \$nnnn	42 42 4E	5	13	<i>dmr</i>		
base-page quad,X	LSRQ \$nn,X	42 42 56	4	12	<i>dmpr</i>		
absolute quad,X	LSRQ \$nnnn,X	42 42 5E	5	13	<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.

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, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

ORA : Binary OR

45GS02

 $A \leftarrow A \text{ OR } M$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles
[indirect],Z	ORA [\$nn],Z	EA 12	3	7 <i>ipr</i>

i Add one cycle if clock speed is at 40 MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.**ORQ**

This instruction performs a binary OR operation of the argument with the Q pseudo register, and stores the result in the Q pseudo register. Only bits that were already set in the Q pseudo register, or that are set in the argument, or both, will be set in the Q pseudo register on completion.

Note that this operation treats the memory address as the first address of a 32-bit little-endian value. That is, the memory address and the three following will be used.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.

ORQ : Binary OR Quad

45GS02

 $Q \leftarrow Q \text{ OR } M$

N	Z	I	C	D	V	E
+	+

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page quad	ORQ \$nn	42 42 05	4	8 <i>r</i>
absolute quad	ORQ \$nnnn	42 42 0D	5	9 <i>r</i>
(indirect quad)	ORQ (\$nn)	42 42 12	4	10 <i>pr</i>
[indirect quad]	ORQ [\$nn]	42 42 EA 12	5	13 <i>pr</i>

p Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

RESQ

These extended opcodes are reserved, and their function is undefined and subject to change in future revisions of the 45GS02. They should therefore not be used in any program.

RESQ : Reserved extended opcode		45GS02				
UNDEFINED		N Z I C D V E				
Addressing Mode	Assembly	Code	Bytes	Cycles		
(indirect quad,X)	RESQ (\$nn,X)	42 42 01	4	10	<i>ipr</i>	
(indirect quad),Y	RESQ (\$nn),Y	42 42 11	4	10	<i>ipr</i>	
base-page quad,X	RESQ \$nn,X	42 42 15	4	8	<i>pr</i>	
absolute quad,Y	RESQ \$nnnn,Y	42 42 19	5	9	<i>pr</i>	
absolute quad,X	RESQ \$nnnn,X	42 42 1D	5	9	<i>pr</i>	
(indirect quad,X)	RESQ (\$nn,X)	42 42 21	4	10	<i>ir</i>	
(indirect quad),Y	RESQ (\$nn),Y	42 42 31	4	10	<i>ipr</i>	
base-page quad,X	RESQ \$nn,X	42 42 34	4	8	<i>pr</i>	
base-page quad,X	RESQ \$nn,X	42 42 35	4	8	<i>pr</i>	
absolute quad,Y	RESQ \$nnnn,Y	42 42 39	5	10	<i>pr</i>	
absolute quad,X	RESQ \$nnnn,X	42 42 3C	5	9	<i>pr</i>	
absolute quad,X	RESQ \$nnnn,X	42 42 3D	5	10	<i>pr</i>	
(indirect quad,X)	RESQ (\$nn,X)	42 42 41	4	10	<i>ipr</i>	
(indirect quad),Y	RESQ (\$nn),Y	42 42 51	4	10	<i>ipr</i>	
base-page quad,X	RESQ \$nn,X	42 42 55	4	8	<i>pr</i>	
absolute quad,Y	RESQ \$nnnn,Y	42 42 59	5	9	<i>pr</i>	
absolute quad,X	RESQ \$nnnn,X	42 42 5D	5	9	<i>pr</i>	
(indirect quad,X)	RESQ (\$nn,X)	42 42 61	4	10	<i>ir</i>	
(indirect quad),Y	RESQ (\$nn),Y	42 42 71	4	10	<i>ipr</i>	
base-page quad,X	RESQ \$nn,X	42 42 75	4	8	<i>pr</i>	
absolute quad,Y	RESQ \$nnnn,Y	42 42 79	5	10	<i>pr</i>	
absolute quad,X	RESQ \$nnnn,X	42 42 7D	5	10	<i>pr</i>	

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

p Add one cycle if indexing crosses a page boundary.

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

ROLQ

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 31 will be shifted out into the Carry Flag.

NOTE: The memory address is treated as the first address of a little-endian encoded 32-bit value.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The C flag will be set if bit 31 of the value was set, prior to being shifted, otherwise it will be cleared.

ROLQ : Rotate Left Quad		45GS02					
$M \leftarrow M \ll 1, C \leftarrow M(31), M(0) \leftarrow C$							
		N Z I C D V E					
		+ + . + . . .					
Addressing Mode	Assembly	Code	Bytes	Cycles			
base-page quad	ROLQ \$nn	42 42 26	4	12	<i>dmr</i>		
Q Pseudo Register	ROLQ Q	42 42 2A	3	3			
absolute quad	ROLQ \$nnnn	42 42 2E	5	13	<i>dmr</i>		
base-page quad,X	ROLQ \$nn,X	42 42 36	4	12	<i>dmpr</i>		
absolute quad,X	ROLQ \$nnnn,X	42 42 3E	5	13	<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.

RORQ

This instruction shifts either the Q pseudo register or contents of the provided memory location one bit right. Bit 31 will be set to the current value of the Carry Flag, and the bit 0 will be shifted out into the Carry Flag.

Note that the address is treated as the first address of a little-endian 32-bit value.

Side effects

- The N flag will be set if the result is negative, i.e. has its most significant bit set, otherwise it will be cleared.
- The Z flag will be set if the result is zero, otherwise it will be cleared.
- The C flag will be set if bit 31 of the value was set, prior to being shifted, otherwise it will be cleared.

RORQ : Rotate Right Quad		45GS02			
M $\leftarrow M \gg 1$, C $\leftarrow M(0)$, M(31) $\leftarrow C$					
		N Z I C D V E			
		+ + . . + . . .			
Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page quad	RORQ \$nn	42 42 66	4	12	<i>dmr</i>
Q Pseudo Register	RORQ Q	42 42 6A	3	3	
absolute quad	RORQ \$nnnn	42 42 6E	5	13	<i>dmr</i>
base-page quad,X	RORQ \$nn,X	42 42 76	4	12	<i>dmpr</i>
absolute quad,X	RORQ \$nnnn,X	42 42 7E	5	13	<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.

RSVQ

These extended opcodes are reserved, and their function is undefined and subject to change in future revisions of the 45GS02. They should therefore not be used in any program.

UNDEFINED

N Z I C D V E

Addressing Mode	Assembly	Code	Bytes	Cycles	
(indirect quad,X)	RSVQ (\$nn,X)	42 42 81	4	10	<i>ip</i>
(indirect quad,SP),Y	RSVQ (\$nn,SP),Y	42 42 82	4	10	<i>ip</i>
(indirect quad),Y	RSVQ (\$nn),Y	42 42 91	4	10	<i>ip</i>
base-page quad,X	RSVQ \$nn,X	42 42 95	4	8	<i>p</i>
absolute quad,Y	RSVQ \$nnnn,Y	42 42 99	5	9	<i>p</i>
absolute quad,X	RSVQ \$nnnn,X	42 42 9D	5	9	<i>p</i>
(indirect quad,X)	RSVQ (\$nn,X)	42 42 A1	4	10	<i>ipr</i>
(indirect quad,X)	RSVQ (\$nn,X)	42 42 C1	4	10	<i>ipr</i>
(indirect quad),Y	RSVQ (\$nn),Y	42 42 D1	4	10	<i>ipr</i>
base-page quad,X	RSVQ \$nn,X	42 42 D5	4	8	<i>pr</i>
absolute quad,Y	RSVQ \$nnnn,Y	42 42 D9	5	9	<i>pr</i>
absolute quad,X	RSVQ \$nnnn,X	42 42 DD	5	9	<i>pr</i>
(indirect quad,X)	RSVQ (\$nn,X)	42 42 E1	4	10	<i>ipr</i>
(indirect quad),Y	RSVQ (\$nn),Y	42 42 F1	4	10	<i>ipr</i>
base-page quad,X	RSVQ \$nn,X	42 42 F5	4	8	<i>pr</i>
absolute quad,Y	RSVQ \$nnnn,Y	42 42 F9	5	8	<i>pr</i>
absolute quad,X	RSVQ \$nnnn,X	42 42 FD	5	9	<i>pr</i>

i Add one cycle if clock speed is at 40 MHz.*p* Add one cycle if indexing crosses a page boundary.*r* Add one cycle if clock speed is at 40 MHz.

SBC

This instruction performs $A - M - 1 + C$, and sets the processor flags accordingly. The result is stored in the Accumulator Register.

NOTE: 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 of $A - M$ is negative, i.e. has its most significant bit set, otherwise 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, otherwise it will be cleared.

- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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, otherwise it will be cleared.

SBC : Subtract With Carry		45GS02		
$A \leftarrow -M - 1 + C$				
Addressing Mode	Assembly	Code	Bytes	Cycles
[indirect],Z	SBC [\$nn],Z	EA F2	3	0

SBCQ

This instruction performs $Q - M - 1 + C$, and sets the processor flags accordingly. The result is stored in the Q pseudo register.

NOTE: the indicated memory location is treated as the first byte of a 32-bit little-endian value.

NOTE: If the D flag is set, the operation is undefined and subject to change.

Side effects

- The N flag will be set if the result of $A - M$ is negative, i.e. has its most significant bit set, otherwise 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, otherwise it will be cleared.
- The V flag will be set if the result has a different sign to both of the arguments, otherwise 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, otherwise it will be cleared.

SBCQ : Subtract With Carry Quad

45GS02

$$Q \leftarrow Q - M - 1 + C$$

N	Z	I	C	D	V	E
+	+	.	+	.	+	.

Addressing Mode	Assembly	Code	Bytes	Cycles	
base-page quad	SBCQ \$nn	42 42 E5	4	8	<i>r</i>
absolute quad	SBCQ \$nnnn	42 42 ED	5	9	<i>r</i>
(indirect quad)	SBCQ (\$nn)	42 42 F2	4	10	<i>ipr</i>
[indirect quad]	SBCQ [\$nn]	42 42 EA F2	5	13	<i>ipr</i>

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

p Add one cycle if indexing crosses a page boundary.

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

STA

This instruction stores the contents of the Accumulator Register into the indicated location.

STA : Store Accumulator

45GS02

$$M \leftarrow A$$

N	Z	I	C	D	V	E
.

Addressing Mode	Assembly	Code	Bytes	Cycles	
[indirect],Z	STA [\$nn],Z	EA 92	3	8	<i>ip</i>

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

p Add one cycle if indexing crosses a page boundary.

STQ

This instruction stores the contents of the Q pseudo register into the indicated location.

As Q is composed of A, X, Y and Z, this means that these four registers will be written to the indicated memory location through to the indicated memory location plus 3, respectively.

STQ : Store Q**45GS02**M \leftarrow A, M+1 \leftarrow X, M+2 \leftarrow Y, M+3 \leftarrow Z**N Z I C D V E**

Addressing Mode	Assembly	Code	Bytes	Cycles
base-page quad	STQ \$nn	42 42 85	4	8
absolute quad	STQ \$nnnn	42 42 8D	5	9
(indirect quad)	STQ (\$nn)	42 42 92	4	10 <i>ip</i>
[indirect quad]	STQ [\$nn]	42 42 EA 92	5	13 <i>ip</i>

i Add one cycle if clock speed is at 40 MHz.*p* Add one cycle if indexing crosses a page boundary.

H-146

APPENDIX

Developing System Programmes

- **Introduction**
- **Flash Menu**
- **Format/FDISK Utility**
- **Keyboard Test Utility**
- **MEGA65 Configuration Utility**
- **Freeze Menu**
- **Freeze Menu Helper Programmes**

- Hypervisor
- OpenROM

INTRODUCTION

The MEGA65 has a number of system programs and utilities that are used at various times to perform various functions. This includes the utilities accessible via the Utility Menu , the Freeze Menu and its own helper programs, as well as the Flash Menu .

A number of these system programs are pre-loaded into the MEGA65 bitstream, while others live on the SD card. For those that are pre-loaded into the MEGA65 bitstream, this works by having areas of pre-initialised memory, that contain the appropriate program. For example, the utilities accessible via the Utility Menu are all located in the colour RAM, while the Flash Menu is located at \$50000 - \$57FFF.

In one sense, the easiest way to test new versions of these utilities is to generate a new bitstream with the updated versions. However, synthesising a new bitstream is very time consuming, typically taking an hour on a reasonably fast computer. Therefore this chapter explains the procedure for loading an alternate version of each of these system programs, as well as providing some useful information about these programs, how they operate, and the environment in which they operate compared with normal C64 or C65-mode programs.

FLASH MENU

The flash menu is located in pre-initialised RAM at \$50000 - \$57FFF. It is executed during the first boot each time the MEGA65 is switched on. It is unusual in that it executes in the hypervisor context. This is so that it has access to the QSPI flash, which is not available outside of Hypervisor Mode, so that user programs cannot corrupt the cores stored in the flash.

It is also important to note that the flash menu program must fit *entirely* below \$8000 when loaded *and* executing, as the Hypervisor is still mapped at \$8000 - \$BFFF, and can easily be corrupted by an ill behaved flash menu program. In this regard, the flash menu can be regarded as an extension of the hypervisor that is discarded after the first boot. This is unlike all other system programs, that operate in a dedicated memory context, from where the Hypervisor is safe from corruption. It also means that you can't crunch the flash menu to make it fit, as it would overwrite the Hypervisor during de-crunching.

Also, as the flash menu is executed very early in the boot process, only the pre-included OpenROM ROM image is available. Thus you must ensure that your flash menu program is compatible with that ROM.

The Hypervisor maintains a flag that indicates whether the flash menu has been executed or not. This flag is updated at the point where the Hypervisor exits to user

mode for the first time, since after that point, the contents of \$50000 - \$57FFF can no longer be trusted to contain the flash menu. This means that if you wish to have the Hypervisor run a new version of the flash menu that you have loaded, you must prevent the Hypervisor from exiting to user mode first.

The easiest way to achieve this is to hold the ALT key down while powering on the MEGA65. This will cause the Hypervisor to display the Utility Menu, rather than exiting to user mode. It is safe at this time to use the `m65` utility to load the replacement flash menu program using a command similar to the following:

```
m65 -@ newflashmenu.prg@50000
```

That command would load the file `newflashmenu.prg` at memory location \$50000.

After that, you can simply press the reset button on the side of the MEGA65 while holding **NO SCROLL** down, and it will boot again, and because it never left Hypervisor Mode during the previous boot cycle, it will run your updated flash menu program.

It should also be possible to completely automate this process, by first using `m65 -b` to load a new bitstream, thus simulating a cold boot, and then quickly calling `m65` again to simulate depressing the ALT key (or perhaps simply halting the processor), then `m65 -@ ...` and finally `m65 -F` to reset the machine. Writing a script or utility that correctly implements this automation is left as an exercise for the reader.

FORMAT/FDISK UTILITY

The Format/FDISK utility is accessed as part of the Utility Menu system. These utilities are compiled, crunched and linked using the `utilpacker` program. If you have checked out the `mega65-core` source repository, you can re-build the colour RAM image by using:

```
make bin/COLOURRAM.BIN
```

You will of course need to first have modified the Format/FDISK utility, which is normally located in the `src/mega65-freeze-menu` subdirectory.

You need to then load this modified colour RAM image into the running machine. Similar to when updating the flash menu, the Hypervisor will only present the utility menu on the first boot, before exiting to user mode for the first time, because it cannot otherwise be sure that the colour RAM contains the valid utility programs.

So as for the flash menu, you would power the MEGA65 off, and then holding the ALT key down, you switch the MEGA65 back on, so that it displays the utility menu. At this point you can use the following command to load your modified COLOURRAM.BIN file:

```
m65 -c COLOURRAM.BIN
```

You can now hold **ALT** down, and press the reset button on the left-hand side of the MEGA65, which should again present the utility menu, but this time with your modified format/fdisk utility in place.

KEYBOARD TEST UTILITY

The process for updating the Keyboard test utility is essentially the same as for the format/FDISK utility, as it lives in the colour RAM

MEGA65 CONFIGURATION UTILITY

The process for updating the MEGA65 Configuration utility is essentially the same as for the format/FDISK utility, as it lives in the colour RAM

FREEZE MENU

The Freeze Menu is a normal program, which is stored in FREEZER.M65 on the SD card's FAT32 file system.

To updated the Freeze Menu, simply use the m65ftp utility or some other means to upload your updated FREEZER.M65 file to the SD card's FAT32 file system. The format of the program is simply a C64-mode PRG file, just renamed to FREEZER.M65.

FREEZE MENU HELPER PROGRAMMES

The Freeze Menu helper programs are updated in the same way as the Freeze Menu itself.

HYPERVISOR

The Hypervisor is normally built as HICKUP.M65, a 16KB file that contains the complete Hypervisor program. MEGA65 bitstreams contain a pre-build version located at \$FFF8000 - \$FFFFBFFF. Updated versions of the Hypervisor can be tested using two main approaches:

- 1. Place the updated HICKUP.M65 file on the FAT32 file system of the SD card, and then power the MEGA65 off and on. This works because the Hypervisor contains code that checks for an updated version of itself, and if found, loads it. However this approach is problematic in that if you install a newer bitstream, it will still downgrade the Hypervisor to whatever version is found in the HICKUP.M65 file on the SD card. This method is only recommended for developers who have a need to test their modified Hypervisor code from a cold start. Even then, it is recommended to remove the HICKUP.M65 file immediately after testing to avoid unexpected down-grading in the future.
- 2. Use the m65 command's -k option to replace the Hypervisor in place, and then reset the MEGA65 using the reset button on the left-hand side of the case. This should be done when the Hypervisor is *not* active, so that corruption of current execution cannot occur. However, it must also occur before any ROM has been loaded to replace the default OpenROM image. This is because the Hypervisor will attempt to call into the ROM on first-boot in preparation for calling the flash menu, and assumes that the OpenROM is present, because it uses a special OpenROM-specific call to initialise parts of the system state for the flash menu. This is best done by using a command like `m65 -k bin/HICKUP.M65 -R bin/MEGA65.ROM` to load both a new Hypervisor program and re-load an OpenROM image.

OPENROM

To load a new version of a ROM, there are several options, including replacing both the Hypervisor and ROM at the same time, as described above. However, typically the easiest is to copy the new ROM onto the FAT32 filesystem of the SD card as either MEGA65.ROM, or MEGA65 x .ROM, where x is replaced by a digit between 0 and 9. When resetting the MEGA65, MEGA65.ROM will then be loaded as normal, or if a digit between 0 and 9 is held down on the keyboard while resetting, the Hypervisor will instead load MEGA65 x .ROM, where x is the number being held down on the keyboard.

J APPENDIX

MEGA65 Hyppo Services

- **Introduction**
- **General Services**
- **Drive/Storage Services**
- **Disk Image Services**
- **Task and Process Services**
- **System Partition Services**
- **Freezer Services**

INTRODUCTION

A part of the MEGA65 is the system program called Hypo that:

- Boots the MEGA65.
- Loads the ROMs and other files from the SD card.
- Makes memory banks 2 and 3 ROM-like by protecting them from being written to.
- Virtualises the floppy disk controller so you can use disk images.
- Launches various utilities like the freezer and the Matrix Mode Debugger.
- Provides services specific to the MEGA65 that you can use in your programs.

If you know about hypervisors and virtual machines, Hypo is a very limited hypervisor. Don't expect to be able to run multiple virtual machines concurrently with full isolation. Hypo runs things that are more akin to the task and processes of a modern operating system than the virtual machines of a hypervisor as you might know it.

Hypo provides 3 operating modes.

- **The C64-like operating mode** runs C64 programs and MEGA65 programs that run in the MEGA65's C64 mode. When you boot with  pressed or use the **GO64** command, Hypo starts a process in the C64-like operating mode to run BASIC 2 or the MEGA65 program.
- **The C65-like operating mode** is the MEGA65's normal operating mode. This is where regular MEGA65 program run, including BASIC 65 programs.
- **The MEGA65 operating mode** runs the MEGA65's system programs like the freezer, the configuration utility and the Matrix Mode Debugger. Maybe surprisingly, normal MEGA65 programs do not run in the MEGA65 operating mode. They run in the C65-like operating mode. The MEGA65 operating mode is designed solely for the MEGA65 and does not attempt to be compatible with or even be similar to previous systems.

Unlike on the C128, it is possible for a program to effectively change the operating mode while it's is running, by simply enabling or disabling the various hardware features.

Hippo provides very limited virtualisation of the MEGA65's hardware. It can virtualise the floppy controller. There are plans to virtualise the serial bus so the MEGA65 can use disk images for units like the 1541.

There are some parts of the hardware that only Hippo can access. It is the only component that can directly access the internal and external SD cards. You need to use Hippo's services if you want to access the files and directories on the SD cards from within your programs.

Terminology

When you start to learn about Hippo, there can be some terminology that might be confusing if you already know about other parts of the MEGA65.

On the SD card there is likely to be a file called HICKUP.M65. This file updates Hippo to new versions without having to install an upgraded core. You might find occasions where Hippo might be called Hickup because of this strong association.

There are 3 distinct disk operating systems in the MEGA65.

- Inside Hippo is Hippo DOS, or HDOS for short. HDOS is for accessing the FAT32 file system on the SD cards. HDOS does not know anything about Commodore file systems. It can attach an image of a Commodore file system, but it does not understand what is inside the image.
- Inside the Kernal is CBDOS. CBDOS is for accessing 1581-like file systems. CB-DOS uses the 45IO27 multi-function I/O controller to access the sectors of a physical disk. CBDOS does not know anything about SD cards and the FAT32 file system on them. Hippo virtualises part of the 45IO27 so CBDOS can access disk images like they're physical disks.
- The external disk units attached to the serial bus each have their own DOS. They are used for accessing the file systems on their respective physical disks.

The word drive means different things for each of these DOS's.

- The drives in Hippo are the partitions of the internal and external SD cards. When the MEGA65 boots, Hippo assigns numbers to the partitions it can read.
- The drives in CBDOS are the physical disk drives attached to the 45IO27 multi-function I/O controller — such as the internal disk drive — or the disk images attached to the virtualised 45IO27. The CBDOS drives are normally seen as units 8 and 9.
- The drives in an external unit attached to the serial bus are the disk drives inside that unit.

Versions

This chapter describes the services available in Hyppo 1.2.

New Hyppo services may become available and existing Hyppo services may change or be deprecated. A robust program will use the [hippo_getversion](#) service to check whether it is compatible with the Hyppo in the MEGA65 it's running on.

Using

When you want to use a Hyppo service, you don't use JSR. This is because Hyppo exists in a space that's separate from regular code. In order to access it, the CPU needs to switch into its hypervisor mode.

At addresses \$D640 - \$D67F are a set of hypervisor traps. Writing to these addresses are not like writing to other addresses. Instead of writing to memory or I/O, the CPU switches into the hypervisor mode and starts a Hyppo service. How the CPU does this is described in Chapter/Appendix G on page [G-3](#).

Which Hyppo service starts depends on what value from the A register you write and which trap you write to. Each of the services described in this chapter tells you what value to write and which trap to use. You have to use the A register when triggering a trap. Writing the same value from another register won't work.

When the Hyppo service finishes, the CPU will switch back to your program. Except for the registers a service uses to return values, the registers are otherwise preserved.

Important The CPU may or may not execute the next byte in your program after the Hyppo service finishes. Put a CLV instruction after the STA. The CPU executing the CLV or not shouldn't matter to your program. If your program does rely on the V flag, you can use the NOP instruction instead. When you use NOP you must be mindful of when the CPU interprets the NOP as a prefix for the following instruction. For this reason, you should prefer using CLV over NOP.

Errors

If the service was successful, it will set the C flag.

If the service was unsuccessful, it will clear the C flag and put an error code in the A register. There is a table of error codes in the description for [hippo_geterrorcode](#).

Examples

The examples use the ACME assembler. The ACME assembler is not required. The Hippo services can be used with any assembler.

The examples are often not complete. They assume an error handler called `error` is defined somewhere. They also assume a transfer area has been defined somewhere. [hippo_setup_transfer_area](#) and [hippo_setname](#) show how to define a transfer area.

GENERAL SERVICES

hippo_geterrorcode

- Trap:** LDA #\$38 : STA \$D643 : CLV
- Service:** Returns the current error code from Hippo.
- Precondition:** The previous service used cleared the C flag.
- Outputs:** A The error code of the previously failed service.
- History:** Available since Hippo 1.2
- Remarks:** The error code is only valid if the previous Hippo service cleared the C flag. If the C flag was set there was no error and the Hippo error code is undefined.
- The meanings here are generic. See the sections for the services for more specific meanings.
- Error codes:** This is possibly not an exhaustive list.

Hex	Dec	Name	General meaning
\$01	1	partition not interesting	The partition is not of a supported type.
\$02	2	bad signature	The signature bytes at the end of a partition table or of the first sector of a partition were missing or incorrect.
\$03	3	is small FAT	This is partition is FAT12 or FAT16 partition. Only FAT32 is supported.
\$04	4	too many reserved clusters	The partition has more than 65,535 reserved sectors.
\$05	5	not two FATs	The partition does not have exactly two copies of the FAT structure.
\$06	6	too few clusters	The partition contains too few clusters.
\$07	7	read timeout	It took to long to read from the SD card.
\$08	8	partition error	An unspecified error occurred while handling a partition.
\$10	16	invalid address	An invalid address was supplied in an argument.

Hex	Dec	Name	General meaning
\$11	17	illegal value	An illegal value was supplied in an argument.
\$20	32	read error	An unspecified error occurred while reading.
\$21	33	write error	An unspecified error occurred while writing.
\$80	128	no such drive	The supplied Hyppo drive number does not exist.
\$81	129	name too long	The supplied filename was too long.
\$82	130	not implemented	The Hyppo service is not implemented.
\$83	131	file too long	The file is larger than 16MB.
\$84	132	too many open files	All of the file descriptors are in use.
\$85	133	invalid cluster	The supplied cluster number is invalid.
\$86	134	is a directory	An attempt was made to operate on a directory, where a normal file was expected.
\$87	135	not a directory	An attempt was made to operate on a normal file, where a directory was expected.
\$88	136	file not found	The file could not be located in the current directory of the current drive.
\$89	137	invalid file descriptor	An invalid or closed file descriptor was supplied.
\$8A	138	image wrong length	The disk image file has the wrong length.
\$8B	139	image fragmented	The disk image is not stored contiguously on the SD card.
\$8C	140	no space	The SD card has no free space for the requested operation.
\$8D	141	file exists	A file already exists with the given name.
\$8E	142	directory full	The directory cannot accommodate any more entries.
\$FF	255	eof	The end of a file or directory was encountered.
\$FF	255	no such trap	There is no Hyppo service available for the trap. The program may be incompatible with this version of Hyppo.

hippo_getversion

Trap:	LDA #\$00 : STA \$D640 : CLV
Service:	Returns the version of Hippo A.X and HDOS Y.Z .
Outputs:	A The major version number of Hippo X The minor version number of Hippo Y The major version number of HDOS Z The minor version number of HDOS
History:	Available since Hippo 1.2
Remarks:	The HDOS in Hippo is not related to the CBDOS inside the Kernel or the DOS in the disk drive units attached to the serial port.
Example:	Tests if Hippo's version is \geq 1.2 and $<$ 2.0.

```
; Get the version numbers
LDA #$00 : STA $D640 : CLV
; Test if the major version number of Hippo in A is 1
CMP #1 : BNE incompatible
; Test if the minor version number of Hippo in X is 2 or more
TXA : CMP #2 : BMI incompatible
; This Hippo version is compatible
```

hippo_setup_transfer_area

Trap:	LDA #\$3A : STA \$D640 : CLV
Service:	Sets up the area Hypo uses to transfer data to and from your program.
Inputs:	Y The MSB of the transfer area's address
Errors:	\$10 invalid address The transfer area address in Y > \$7E
History:	Available since Hypo 1.2
Remarks:	The transfer area must be between \$0000 and \$7E00. It must also begin on a page boundary. The LSB of its address must be \$00. The transfer area is 256 bytes long for most services. The transfer area is indicated using the CPU's current memory mapping at the time that a service is used. However, it is good practice to always place it in the bottom 32KB of bank 0.
Example:	Reserves 256 bytes on a page boundary and sets it up as the transfer area.

```
;-----  
; Somewhere in the program's data area  
  
; Align with the new page boundary  
!align 255, 0  
  
transferarea:  
; Reserve 256 bytes  
!skip 256  
  
;-----  
; Elsewhere in the program's code  
  
setuptransferarea:  
; Set the transferarea as the Hypo transfer area  
LDY #>transferarea : LDA #$3A : STA $D640 : CLV : BCC error
```

DRIVE/STORAGE SERVICES

In Hyppo, drives are the partitions of the internal and external SD cards. They are not the drive 0 and drive 1 of the F011 floppy controller. They are also not the drive 0 and drive 1 of dual-drive units attached to the serial bus.

hippo_chdir

Trap: LDA #\$0C : STA \$D640 : CLV

Service: Changes the current working directory.

Preconditions: The FAT dir entry for the directory you want to change to has been found. `hippo_findfile` is typically used to find a FAT dir entry. `hippo_findfirst`, `hippo_findnext` and `hippo_readdir` can also be used.

Errors: **\$87 not a directory** The FAT dir entry last found isn't for a directory. Bit 4 of the FAT dir entry's attribute byte is set for directories.

History: Available since Hyppo 1.2

Remarks: You can move up to the parent directory by finding the .. FAT dir entry.

You cannot move up or down more than one directory at a time.

Use `hippo_cdrootdir` to directly change back to the root directory.

Example: Changes to an arbitrary path on the current drive. Call with Y:X being the address of the path. The last character of each component in the path needs to have bit 7 set. The whole path is terminated with a \$00. For example, to change into DIR1 and then DIR2 the path would be !text "DIR", '1'+\$80, "DIR" '2'+\$80, 0.

If successful, returns with the C flag set. If some part of the path doesn't exist, returns with the C flag cleared and the current working directory will be whatever directory was last successfully navigated to.

```
!addr pathptr = $C4
chdirpath:
    STX pathptr : STY pathptr1 ; Set pathptr to Y:X
    Bchdirpath10:
```

... continues on the next page ...

```

JSR &copycomponent ; Get the next component of the path
BEQ &chdirpath20 ; Stop at a 0 byte in the path
JSR &trychdir ; Try to change into the directory
BCC &chdirpath30 ; Stop if we cannot change into the directory
BRA &chdirpath10 ; Repeat with the next component
&chdirpath20: SEC
&chdirpath30: RTS
-----
&trychdir:
    ; Set the Hyppo filename from transferbuffer
    LDY #>transferbuffer : LDA #$2E : STA $D640 : CLV : BCC &trychdir10
    ; Find the FAT dir entry
    LDA #$34 : STA $D640 : CLV : BCC &trychdir10
    ; Chdir into the directory
    LDA #$0C : STA $D640 : CLV
&trychdir10:
    RTS
-----
    ; Copy the next component of the path into transferbuffer
&copycomponent:
    LDY #0
&copycomponent10:
    LDA (pathptr),Y
    BEQ &copycomponent20
    TAX
    AND #$7F : STA transferbuffer,Y
    IMW
    TXA : BPL &copycomponent10
    ; Add a 0 terminating byte to transferbuffer
    LDA #0 : STA transferbuffer,Y
    ; Move pathptr
    TYA
    CLC : ADC pathptr : STA pathptr : LDA #0 : ADC pathptr+1 : STA pathptr+1
&copycomponent20:
    RTS

```

hippo_cdrootdir

Trap: LDA #\$3E : STA \$D643 : CLV

Service: Sets the current directory to the root directory.

Precondition: None.

Outputs: A Any error code that can be returned by ??

History: Available since Hyppo 1.16

hippo_closeall

Trap: LDA #\$22 : STA \$D640 : CLV

Service: Closes all the file descriptors.

Postconditions: Using any file descriptor with `hippo_closedir` or `hippo_closefile` succeeds.

Using any file descriptor with `hippo_readdir` or `hippo_readfile` fails.

`hippo_opendir` and `hippo_openfile` reuse the file descriptor.

History: Available since Hippo 1.2

Remarks: You can also close individual file descriptors using `hippo_closedir` or `hippo_closefile`.

hippo_closedir

Trap: LDA ##\$16 : STA \$D640 : CLV

Service: Closes a file descriptor for a directory.

Preconditions: The file descriptor given in the X register was opened using [hippo_opendir](#) .

Inputs: X The file descriptor for the directory

Postconditions: Using the file descriptor again with [hippo_closedir](#) succeeds.

Using the file descriptor again with [hippo_readdir](#) fails.

[hippo_opendir](#) and [hippo_openfile](#) reuse the file descriptor.

History: Available since Hippo 1.2

Remarks: You can also close all the open file descriptors using [hippo_closeall](#) .

Example: See the example in [hippo_opendir](#) .

hippo_closefile

Trap: LDA #\$20 : STA \$D640 : CLV

Service: Closes a file descriptor for a file.

Preconditions: The file descriptor given in the X register was opened using `hippo_openfile`.

Inputs: X The file descriptor for the file

Postconditions: Using the file descriptor again with `hippo_closefile` succeeds.

Using the file descriptor again with `hippo_readfile` fails.

`hippo_opendir` and `hippo_openfile` reuse the file descriptor.

History: Available since Hyppo 1.2

Remarks: You can also close all the open file descriptors using `hippo_closeall`

hippo_filedate

Trap: LDA #\\$2C : STA \\$D640 : CLV

Service: Sets time stamp of a file.

Remarks: **NOT IMPLEMENTED**

hippo_findfile

Trap: LDA ##\$34 : STA \$D640 : CLV

Service: Finds the first file whose filename matches the current Hippo filename.

Preconditions: The current Hippo filename has been set using [hippo_setname](#) .

Postconditions: No additional file descriptors are open.

Errors: **\$88 file not found** A matching file was not found in the current directory of the current drive.

History: Available since Hippo 1.2

Remarks: Hippo will only find files whose long filename is all in uppercase. Hippo converts the current filename to ASCII uppercase before trying to match it. Bytes \$61 - \$7B change to \$41 - \$5A.

Hippo does not yet support the wildcard characters * and ?. Support for that is planned in a future version.

This only finds the first matching file. You can find multiple matches by using [hippo_findfirst](#) and [hippo_findnext](#) .

Example: See the example in [hippo_openfile](#) .

hippo_findfirst

Trap:	LDA #\$30 : STA \$D640 : CLV
Service:	Finds the first file whose filename matches the current Hippo filename.
Preconditions:	The current Hippo filename has been set using hippo_setname .
Outputs:	A The file descriptor for reading the current working directory. You might be responsible for closing this file descriptor using hippo_closedir . See the remarks.
Postconditions:	hippo_findnext find the next matching file or fails with a file not found error.
Side effects:	Sets the current file descriptor.
Errors:	\$88 file not found A matching file was not found in the current directory of the current drive.
History:	Available since Hippo 1.2
Remarks:	If Hippo finds an initial matching file, it will set the C flag and return a file descriptor in the A register. This is a file descriptor for reading the current working directory. You are responsible for closing it using hippo_closedir . It's a standard directory file descriptor. You can use hippo_readdir to read the FAT dir entries after the file that was found. If Hippo doesn't find any matching files, it will fail with a file not found error. In this case Hippo will have already closed the file descriptor and you don't have to close it. Hippo will only find files whose long filename is all in uppercase. Hippo converts the current filename to ASCII uppercase before trying to match it. Bytes \$61 - \$7B change to \$41 - \$5A. Hippo does not yet support the wildcard characters * and ?. Support for that is planned in a future version. If you are only interested in the first match, you can use hippo_findfile instead. hippo_findfile always closes the file descriptor for you. But you can't use it to find multiple matching files.
Example:	See the example in hippo_findnext .

hippo_findnext

- Trap:** LDA #\$32 : STA \$D640 : CLV
- Service:** Finds a subsequent file whose filename matches the current Hippo filename.
- Preconditions:** The current Hippo filename has been set using [hippo_setname](#).
The first matching file has already been found successfully using [hippo_findfirst](#).
- Postconditions:** Using [hippo_findnext](#) again finds the next matching file or fails with a file not found error.
- Errors:** **\$88 file not found** A subsequent matching file was not found in the current directory of the current drive.
- History:** Available since Hippo 1.2
- Remarks:** If Hippo doesn't find a subsequent matching file, it will fail with a file not found error. Hippo will also close the file descriptor it output in [hippo_findfirst](#).
If you don't exhaust the search by using [hippo_findnext](#) until it fails with a file not found error, you are required to close the file descriptor yourself using [hippo_closedir](#).
- Example:** Returns with X register containing the number of files matching the current Hippo filename in the current working directory of the current drive. While a directory can in theory have multiple files with an identical name, this example will be more useful once Hippo supports * and ? wildcards.

```
; Assume the current Hippo filename has already been set.  
; Reset the count in X.  
LDX #0  
; Try to find the first matching file.  
LDA #$30  
C10:  
STA $D640 : CLV : BCC C20  
; Increment the count in X  
INX  
; Try to find the next matching file.  
LDA #$32  
BRA C10
```

... continues on the next page ...

```
020:  
; If the error code in A is $88 there are no more matching files,  
; otherwise an error occurred.  
CMP #$88 : BNE error  
; No need to close the file handle because the search exhausted.
```

hippo_fstat

Trap: LDA #\$28 : STA \$D640 : CLV

Service: Returns information about a file.

Remarks: **NOT IMPLEMENTED**

hippo_getcurrentdrive

Trap:	LDA #\$04 : STA \$D640 : CLV
Service:	Returns the number of the currently selected drive (SD card partition).
Outputs:	A The current drive number
History:	Available since Hyppo 1.2
Remarks:	hippo_selectdrive changes the current drive number. hippo_cdrootdir can also change it.
Example:	Prints the number of the currently selected drive in the top-left of the screen. This example assumes that there aren't more than 10 drives (drives 0 to 9). It also assumes the screen memory hasn't been moved from \$800.

```
; Get the current drive  
LDA #$04 : STA $D640 : CLV : BCC error  
  
; Convert the drive number in A into a screen code  
CLC : ADC #$30  
  
; Put the screen code into the top-left of screen memory  
STA $800
```

hippo_getcwd

Trap: LDA #\$0A : STA \$D640 : CLV

Service: Returns information on the currently selected directory or sub-directory.

Remarks: **NOT IMPLEMENTED**

hippo_getdefaultdrive

Trap: LDA #\$02 : STA \$D640 : CLV

Service: Returns the drive number (SD card partition) Hippo selected while booting.

Outputs: A The default drive number

History: Available since Hippo 1.2

Example: Selects the default drive.

```
; Get the default drive  
LDA #$02 : STA $D640 : CLV : BCC @error  
  
; Transfer the drive number in A to X  
TAX  
  
; Select the default drive  
LDA #$06 : STA $D640 : CLV : BCC @error
```

hippo_getdrivesize

Trap: LDA #\$08 : STA \$D640 : CLV

Service: Returns information on the size of the currently selected drive (SD card partition).

Remarks: **NOT IMPLEMENTED**

hippo_loadfile

Trap:	LDA #\$36 : STA \$D640 : CLV
Service:	Loads a file into chip memory.
Preconditions:	The name of the file to load has been set using hippo_setname .
Inputs:	X The LSB of the address to start loading from Y The middle byte of the address to start loading from Z The MSB of the address to start loading from
Postconditions:	No additional file descriptors are open.
Errors:	\$84 too many open files hippo_loadfile uses one file descriptor internally, but all the file descriptors are in use. Close some or all of the file descriptors using hippo_closedir , hippo_closefile or hippo_closeall . \$88 file not found The file was not found in the current directory of the current drive.
History:	Available since Hypo 1.2
Remarks:	This service can load files up to 16MB in size into the first 16MB of chip memory. Chip memory is the 384KB or more of memory inside the CPU module. Loading will start at 28-bit address \$00ZZYYYYXX. If loading tries to go beyond \$00FFFFFF, it wraps around and continue at \$00000000. You can use hippo_loadfile_attic to load a file into hyper memory. The hyper memory is the 8MB or more of memory in the external RAM chips.
Example:	Loads a file into memory starting at \$48000.

```
; Assume the current Hypo filename has already been set.  
; No need to find the file first.  
LDZ #$04 ; Most significant byte  
LDY #$00 ; Middle byte  
LDX #$00 ; Least significant byte  
LDA #$36 : STA $D640 : CLV : BCC error
```

hippo_loadfile_attic

Trap:	LDA #\$3E : STA \$D640 : CLV
Service:	Loads a file into hyper memory.
Preconditions:	The name of the file to load has been set using hippo_setname .
Inputs:	X The LSB of the address to start loading from Y The middle byte of the address to start loading from Z The MSB of the address to start loading from
Postconditions:	No additional file descriptors are open.
Errors:	\$84 too many open files hos_loadfile_attic uses one file descriptor internally, but all the file descriptors are in use. Close some or all of the file descriptors using hippo_closedir , hippo_closefile or hippo_closeall . \$88 file not found The file was not found in the current directory of the current drive.
History:	Available since Hypo 1.2
Remarks:	This service can load files up to 16MB in size into the first 16MB of hyper memory. Hyper memory is the 8MB or more of memory in the external RAM chips. Loading will start at 28-bit address \$08ZZYYXX. If loading tries to go beyond \$08FFFFFF, the loading will wrap around and continue at \$08000000. You can use hippo_loadfile to load a file into chip memory. The chip memory is the 384KB or more of memory inside the CPU module.

hippo_mkdir

Trap: LDA #\$0E : STA \$D640 : CLV

Service: Creates a sub-directory.

Errors: **\$8D file exists** A sub-directory or file already exists with the current Hippo filename in the current working directory of the current drive.

Remarks: NOT IMPLEMENTED

hippo_mkfile

Trap: LDA #\$1E : STA \$D640 : CLV

Service: Creates a file.

Errors: **\$8D file exists** A sub-directory or file already exists with the current Hippo filename in the current working directory of the current drive.

Remarks: NOT IMPLEMENTED

hippo_opendir

Trap:	LDA ##\$12 : STA \$D640 : CLV
Service:	Opens the current working directory for reading the file entries in it.
Preconditions:	The drive and directory you want to read have already been set up using hippo_selectdrive and hippo_chdir if necessary.
Outputs:	A The file descriptor for reading the directory. You are responsible for closing this file descriptor using hippo_closedir .
Postconditions:	hippo_readdir reads the first FAT dir entry in the directory.
Errors:	\$84 too many open files All the file descriptors are in use. hippo_opendir and hippo_openfile share the same very small pool of file descriptors. Close some or all of the file descriptors using hippo_closedir , hippo_closefile or hippo_closeall .
	\$87 not a directory The FAT dir entry last found is for a file. Use hippo_openfile for files.
History:	Available since Hippo 1.2
Example:	Calls processdirentry for each FAT dir entry in the current working directory. processdirentry is assumed to be defined elsewhere.

```
; Open the current working directory
LDA ##$12 : STA $D640 : CLV : BCC error
; Transfer the directory file descriptor into X
TAX
; Set Y to the MSB of the transfer area
LDY ##>transferarea
C10:
; Read the directory entry
LDA ##$14 : STA $D640 : CLV : BCC C20
; Call processdirentry (assumed to be defined elsewhere)
PHX : PHY : JSR processdirentry : PLY : PLX
BRA C10
C20:
; If the error code in A is $85 we have reached the end of the
; directory otherwise there's been an error
CMP ##$85 : BNE error
; Close the directory file descriptor in X
LDA ##$16 : STA $D640 : CLV : BCC error
```

hippo_openfile

Trap:	LDA ##18 : STA \$D640 : CLV
Service:	Opens a file on a drive.
Preconditions:	The file has already been found. Files can be found using hippo_findfile , hippo_findfirst and hippo_findnext . hippo_readdir can also be used to find a file.
Outputs:	A The file descriptor for accessing the file. You are responsible for closing this file descriptor using hippo_closefile .
Postconditions:	Using hippo_readfile with this file descriptor reads the first sector of the file.
Side effects:	Sets the current file to the newly opened file. hippo_readfile reads from the current file.
Errors:	\$84 too many open files All the file descriptors are in use. hippo_opendir and hippo_openfile share the same very small pool of file descriptors. Close some or all of the file descriptors using hippo_closedir , hippo_closefile or hippo_closeall .
	\$86 is a directory The FAT dir entry last found is for a directory. Use hippo_opendir for directories.
History:	Available since Hippo 1.2
Remarks:	You cannot use this to open a file inside a disk image. To do that you use hippo_d81attach0 or hippo_d81attach1 to attach the disk image and then use either use the Kernal to read the file or program the virtualised F011 floppy controller.
Example:	Finds and opens a file.

```
; Assume the current Hippo filename has already been set.  
; Find the file  
LDA ##34 : STA $D640 : CLV : BCC error  
; Open the file  
LDA ##18 : STA $D640 : CLV : BCC error
```

hippo_readdir

Trap: LDA ##\$14 : STA \$D640 : CLV

Service: Reads the next FAT dir entry into a destination area.

Preconditions: The file descriptor given in the X register was opened using `hippo_opendir` and `hippo_closedir` hasn't since been used to close it.

The destination area is on a page boundary between \$0000 and \$7E00 and is at least 87 bytes.

Inputs: **X** The file descriptor for the directory.

Y The MSB of the destination area.

Outputs: Starting at \$YY00, the FAT dir entry has this structure.

Offset	Type	Description
\$00	asciiiz	The long file name
\$40	byte	The length of long file name
\$41	ascii	The "8.3" file name. The name part is padded with spaces to make it exactly 8 bytes. The 3 bytes of the extension follow. There is no . between the name and the extension. There is no NULL byte.
\$4E	dword	The cluster number where the file begins. For sub-directories, this is where the FAT dir entries start for that sub-directory.
\$52	dword	The length of file in bytes.
\$56	byte	The type and attribute bits.

This is what the bits in the last byte mean. Bits 6 and 7 are undefined.

Bit	Meaning if bit is set
0	Read only
1	Hidden
2	System
3	Volume label
4	Sub-directory
5	Archive

Postconditions: Using `hippo_readdir` again reads the next FAT dir entry in the directory.

Errors: **\$08 partition error** An unspecified error occurred while handling the currently selected partition.

\$10 invalid address The Y register is > \$7E.

\$85 invalid cluster An attempt was made to read past the end of the directory.

Remarks:

If the long file name in the FAT dir entry is too long to copy into the destination area, Hyppo skips the entry entirely.

The file names in FAT are encoded as UTF-16. Hyppo only reads the LSB of each 16-bit character. Hyppo does not convert file names into PETSCII.

See [hyppo_setup_transfer_area](#) for more details about the value for the Y register.

History:

Available since Hyppo 1.2

Example:

See the example in [hyppo_opendir](#).

hippo_readfile

Trap: LDA #\\$1A : STA \\$D640 : CLV

Service: Reads the next sector of the current file into the sector buffer.

Preconditions: There is a current file open. Files can be opened with [hippo_openfile](#).

Outputs: **X** The LSB of the number of bytes read

Y The MSB of the number of bytes read

Postconditions: The next call to [hippo_readfile](#) will read the next sector of the current file or signal the end of the file.

Errors: **\$89 invalid file descriptor** There is no current file.

History: Available since Hypno 1.2

Remarks: To access the data, you need to either:

- map the sector buffer into the 16-bit address space;
- use an enhanced DMA transfer to copy the sector buffer at \$FFD6E00 - \$FFD6FFF into a buffer already mapping into the 16-bit address space; or
- use 32-bit load instructions to access the sector buffer directly.

If a full sector was read, Y:X will be \$0200. For the last sector of the file, Y:X may be less than that. Any bytes in the sector buffer after Y:X are undefined and will not necessarily be zero.

If you read past the end of the last sector, Y:X will be \$0000, the A register will be \$FF and the C flag will be set.

While multiple files can be opened simultaneously, only the current file can be read. The current file is often the last file opened, but not always.

Example: Maps the sector buffer to \$DE00 and then reads each sector of the file calling processsector for each sector read. processsector is assumed to be defined elsewhere.

```
; Assume the file is already open.  
; Unmap the colour RAM from $DC00 because that will prevent us  
; from mapping in the sector buffer  
LDA $D030 : PHA : AND #$11111110 : STA $D030
```

... continues on the next page ...

```
010:  
    ; Read the next sector  
    LDA #$1A : STA $D640 : CLV : BCC 020  
    ; Map the sector buffer to $DE00  
    LDA #$81 : STA $D680  
    ; Call processsector (assumed to be defined elsewhere)  
    JSR processsector  
    ; Unmap the sector buffer from $DE00  
    LDA #$82 : STA $D680  
    BRA 010  
020:  
    ; If the error code in A is $FF we have reached the end of the file  
    ; otherwise there's been an error  
    CMP #$FF : BNE error  
    ; Map the colour RAM at $DC00 if it was previously mapped  
    PLA : STA $D030
```

hippo_rename

Trap: LDA #\$2A : STA \$D640 : CLV
Service: Renames a file or sub-directory.
Errors: **\$8D file exists** A sub-directory or file already exists with the current Hippo filename in the current working directory of the current drive.
Remarks: **NOT IMPLEMENTED**

hippo_rmdir

Trap: LDA #\$10 : STA \$D640 : CLV
Service: Removes a sub-directory.
Remarks: **NOT IMPLEMENTED**

hippo_rmfile

Trap: LDA #\$26 : STA \$D640 : CLV
Service: Removes a files.
Remarks: **NOT IMPLEMENTED**

hippo_seekfile

Trap: LDA #\$24 : STA \$D640 : CLV
Service: Seeks to a given sector in a file.
Remarks: **NOT IMPLEMENTED**

hippo_selectdrive

- Trap:** LDA #\$06 : STA \$D640 : CLV
- Service:** Sets the currently selected drive (SD card partition).
- Preconditions:** Hippo has assigned a drive number to the SD card partition.
- Inputs:** X The drive number to become the new current drive
- Postconditions:** [hippo_getcurrentdrive](#) returns the value that was in the X register.
Hippo services operate on the newly selected drive.
- Errors:** **\$80 no such drive** The drive in the X register does not exist. Hippo only assigns drive numbers to the SD card partitions it can read.
- History:** Available since Hippo 1.2
- Example:** Tests if drive 2 exists by trying to select it. Returns with the C flag set if drive 2 exists.

```
doesdrive2exist:  
    ; Preserve the current drive so we can restore it later  
    LDA #$04 : STA $D640 : CLV : BCC error  
    PHA  
    ; Try to select drive 2  
    LDX #2 : LDA #$06 : STA $D640 : CLV : BCC 010  
    ; Restore the previously selected drive  
    PLX  
    LDA #$06 : STA $D640 : CLV : BCC error  
    ; The C flag was already set by the Hippo service  
    RTS  
  
010:  
    ; If the error code in A is $80, the drive doesn't exist; otherwise  
    ; some other kind of error occurred  
    CMP #$80  
    BNE error  
    ; Forget about the current drive we preserved because it wasn't  
    ; changed  
    PLX  
    ; Clear the C flag because the drive doesn't exist  
    CLC  
    RTS
```

hippo_setname

- Trap:** LDA #\$2E : STA \$D640 : CLV
- Service:** Sets the current Hippo filename.
- Preconditions:** The filename is stored in ASCII and ends with a \$00 byte.
The filename starts on a page boundary between \$0000 and \$7E00 and is less than 63 characters, excluding the \$00 byte.
- Inputs:** Y The MSB of the filename address.
- Postconditions:** Hippo has copied the filename into its own data area.
The hippo_find* and hippo_load* services use this filename.
- Side effects:** Sets the transfer area to \$YY00.
- Errors:** **\$10 invalid address** The Y register is > \$7E.
\$81 name too long The filename is longer than 63 characters.
- History:** Available since Hippo 1.2
- Remarks:** The filename must be between \$0000 and \$7E00. It must also begin on a page boundary. That is, its address must end with \$00. The current memory mapping is used. However, it is good practice to place it in the bottom 32KB of bank 0.
The filenames in FAT are encoded in UTF-16. Hippo only reads the LSB of each 16-bit character. Hippo does not convert between ASCII and PETSCII.
Hippo accesses the files in the FAT file system on the internal and external SD cards. It does not access files on disks in floppy drives or in disk images.
- Example:** Set the current Hippo filename to GAME.MAP.

```
; Somewhere in the program's data area
!align 255, 0 ; Align with the next page boundary
filename:
!text "GAME.MAP", 0 ; Must end with a 0 byte

; Elsewhere in the program's code
setfilename:
LDY #>filename : LDA #$2E : STA $D640 : CLV : BCC error
```

hippo_writefile

Trap: LDA #\$1C : STA \$D640 : CLV

Service: Writes the sector buffer to the current file.

Remarks: **NOT IMPLEMENTED**

DISK IMAGE SERVICES

The 45IO27 multi-function I/O controller includes a F011-compatible floppy controller. The internal floppy drive is attached to this as drive 0.

Hippo can virtualise the F011 floppy controller so that disk images can be attached instead of floppy drives. Once a disk image is attached, Hippo traps the F011's I/O registers and emulates the commands on the disk image.

You can use BASIC, the Kernal and the F011 I/O registers to operate on a disk image just as you would a physical disk. The virtualisation does not behave the same as a floppy drive in all cases. If you intend for your program to work with both disk images and physical disks, be sure to test it with both.

hippo_d81attach0

Trap: LDA #\$40 : STA \$D640 : CLV

Service: Attach a D81 disk image to virtualised F011 drive 0.

Preconditions: The current Hippo filename has been set using [hippo_setname](#).

Errors: **\$88 file not found** The disk image file was not found in the current directory of the current drive.

History: Available since Hippo 1.2

Remarks: Unless it's been changed, drive 0 of the virtualised F011 floppy controller is unit 8.

Example: Attaches the disk image DISK2.D81 to virtualised F011 drive 0.

```
; Somewhere in the program's data area
!align 255, 0 ; Align with the next page boundary
disk2name:
!text "DISK2.D81", 0 ; Must end with a 0 byte

; Elsewhere in the program's code
attachdisk2:
; Set the Hippo filename to DISK2.D81
LDY #>disk2name : LDA #$2E : STA $D640 : CLV : BCC error
; Attach the disk image
LDA #$40 : STA $D640 : CLV : BCC error
```

hippo_d81attach

Trap: LDA #\$46 : STA \$D640 : CLV

Service: Attach a D81 disk image to virtualised F011 drive 1.

Preconditions: The current Hippo filename has been set using [hippo_setname](#) .

History: Available since Hippo 1.2

Remarks: Unless it's been changed, drive 1 of the virtualised F011 floppy controller is unit 9.

hippo_d81detach

Trap: LDA #\$42 : STA \$D640 : CLV

Service: Detaches any disk images from virtualised F011 drives 0 and 1.

History: Available since Hippo 1.2

hippo_d81write_en

Trap: LDA #\$44 : STA \$D640 : CLV

Service: Enables writing to any disk images attached to virtualised F011 drives 0 and 1.

History: Available since Hippo 1.2

TASK AND PROCESS SERVICES

hippo_create_task_c64

Trap: LDA #\$66 : STA \$D640 : CLV

Service: Creates a Hippo task in the C64-like operating mode.

Remarks: NOT IMPLEMENTED

hippo_create_task_c65

Trap: LDA #\$68 : STA \$D640 : CLV

Service: Creates a Hippo task in the C65-like operating mode.

Remarks: NOT IMPLEMENTED

hippo_create_task_native

Trap: LDA #\$62 : STA \$D640 : CLV

Service: Creates a Hippo task in the MEGA65 operating mode.

Remarks: NOT IMPLEMENTED

hippo_exit_and_switch_to_task

Trap: LDA #\$6A : STA \$D640 : CLV

Service: Exits the current Hippo task and switches context to another Hippo task.

Remarks: NOT IMPLEMENTED

hippo_exit_task

Trap: LDA #\$6E : STA \$D640 : CLV

Service: Exits the current Hippo task.

Remarks: NOT IMPLEMENTED

hippo_get_mapping

Trap: LDA #\$74 : STA \$D640 : CLV

Service: Copies the current 45GS02 memory mapping into a destination area.

Preconditions: The destination area starts on a page boundary between \$0000 and \$7E00 and is at least 6 bytes.

Inputs: Y The MSB of the destination area.

Outputs: Starting at \$YY00, the current mapping info has this structure.

Offset	Type	Description
0	word	MAPLO
2	word	MAPHI
4	byte	The megabyte offset for MAPLO
5	byte	The megabyte offset for MAPHI

Errors: **\$10 invalid address** The Y register is > \$7E.

History: Available since Hippo 1.2

Remarks: MAPLO is the mapping for \$0000 - \$7FFF.

MAPHI is the mapping for \$8000 - \$FFFF.

See Chapter/Appendix G on page G-3 for more information on MEGA65 memory mapping and banking.

hippo_get_proc_desc

Trap: LDA #\$48 : STA \$D640 : CLV

Service: Copies the current task block into a destination area.

Preconditions: The destination area starts on a page boundary between \$0000 and \$7E00 and is at least 256 bytes.

Inputs: Y The MSB of the destination area.

Outputs: Starting at \$YY00, the current task block has this structure.

Offset	Type	Description
\$00	byte	The ID of the current task.
\$01	text	The name of the current task. A maximum of 16 characters. Padded with \$00 bytes. If it's 16 characters, there are no trailing \$00 bytes.
\$11	byte	Flags for the D81 disk image attached to drive 0 of the virtualised F011 floppy controller.
\$12	byte	Same as above but for drive 1.
\$13	byte	The length of the D81 disk image filename attached to drive 0.
\$14	byte	Same as above but for drive 1.
\$15	text	The filename of the D81 disk image attached to drive 0. A maximum of 32 characters. Padded with \$20 bytes. There is no trailing \$00 byte.
\$35	text	Same as above but for drive 1.
\$55		The meaning of these bytes are undefined and subject to change.
\$80		File descriptor 0.
\$A0		File descriptor 1.
\$C0		File descriptor 2.
\$E0		File descriptor 3.

Each of the file descriptors has this structure.

Offset	Type	Description
\$00	byte	The number of the SD card partition where the file resides. \$FF means the file descriptor is closed.
\$01	dword	The cluster where the file starts
\$05	dword	The current cluster
\$09	byte	The current sector within the current cluster
\$0A	dword	The length of the file
\$0E	dword	The current position within the file's buffer
\$12	dword	The cluster of the directory in which the file resides
\$16	word	The index of the file within its directory
\$18	dword	The absolute 32-bit address of the file's buffer
\$1C	word	The number of bytes used in the file's buffer
\$1E	word	The current offset within the file's buffer

Errors:

\$10 invalid address The Y register is > \$7E.

History:

Available since Hyppo 1.2

hippo_gettasklist

Trap: LDA ##\$50 : STA \$D640 : CLV
Service: Gets the list of tasks in Hippo.
Remarks: NOT IMPLEMENTED

hippo_load_into_task

Trap: LDA ##\$64 : STA \$D640 : CLV
Service: Loads a file from an SD card partition into the memory of a Hippo task.
Remarks: NOT IMPLEMENTED

hippo_readoutoftask

Trap: LDA ##\$58 : STA \$D640 : CLV
Service: Reads from the memory of another Hippo task.
Remarks: NOT IMPLEMENTED

hippo_receivemessage

Trap: LDA ##\$54 : STA \$D640 : CLV
Service: Receives messages sent from other Hippo tasks.
Remarks: NOT IMPLEMENTED

hippo_reset

Trap: LDA #\$7E : STA \$D640 : CLV

Service: Warm boots the MEGA65.

History: Available since Hippo 1.2

hippo_rom_writeenable

Trap: LDA #\$02 : STA \$D641 : CLV

Service: Changes \$20000 - \$3FFF to behave like RAM by disabling the write-protection.

History: Available since Hypo 1.2

Remarks: \$20000 - \$3FFF normally has the Kernel, BASIC, CBDOS, and font ROMs.

`hippo_rom_writeprotect` enables the write-protection and blocks writes. `hippo_toggle_rom_writeprotect` toggles the write-protection.

hippo_rom_writeprotect

Trap: LDA #\$00 : STA \$D641 : CLV

Service: Changes \$20000 - \$3FFF to behave like ROM by enabling the write-protection.

History: Available since Hypo 1.2

Remarks: \$20000 - \$3FFF normally has the Kernal, BASIC, CBDOS, and font ROMs.

`hippo_rom_writeenable` disables the write-protection and allows writes. `hippo_toggle_rom_writeprotect` toggles the write-protection.

hippo_sendmessage

Trap: LDA #\$52 : STA \$D640 : CLV

Service: Sends a message to another Hippo task.

Remarks: **NOT IMPLEMENTED**

hippo_serial_monitor_wait_and_write

Trap: LDA #\$xx : STA \$D643 : CLV

Service: Waits for the serial monitor or Matrix Mode Debugger to be ready to receive and then writes a character to it.

Inputs: A The ASCII character to write.

History: Available since Hypo 1.2

Remarks: The service waits for the serial monitor to be ready to receive. This could slow down or hang your program. If you don't want this and you are happy for the character to be lost if the serial monitor is not ready to receive, use [hippo_serial_monitor_write](#).

hippo_serial_monitor_write

Trap: LDA #\$7C : STA \$D640 : CLV

Service: Writes a character to the serial monitor or the Matrix Mode Debugger.

Preconditions: The serial monitor is ready to receive.

Inputs: Y The ASCII character to write.

History: Available since Hippo 1.2

Remarks: The character will be lost if the serial monitor is not ready to receive, If you don't want the character to be lost, use [hippo_serial_monitor_wait_and_write](#).

hippo_set_mapping

Trap: LDA #\$76 : STA \$D640 : CLV

Service: Copies the source area into the current 45GS02 memory mapping.

Preconditions: The source area starts on a page boundary between \$0000 and \$7E00 and is at least 6 bytes.

Inputs: Y The MSB of the source area.

Starting at \$YY00, the current mapping info has this structure.

Offset	Type	Description
0	word	MAPLO
2	word	MAPHI
4	byte	The megabyte offset for MAPLO
5	byte	The megabyte offset for MAPHI

Postconditions: The CPU continues execution with the new memory mapping.

Errors: **\$10 invalid address** The Y register is > \$7E.

History: Available since Hippo 1.2

Remarks: You must take care when changing the memory mapping. Hippo will not take any steps to ensure the instructions after the STA are executed regardless of the mapping. If you change the mapping of the block where the program counter points to, the CPU will resume with the instructions in the newly mapped block.

MAPLO is the mapping for \$0000 - \$7FFF.

MAPHI is the mapping for \$8000 - \$FFFF.

See Chapter/Appendix G on page G-3 for more information on MEGA65 memory mapping and banking.

hippo_switch_to_task

Trap: LDA #\$6C : STA \$D640 : CLV

Service: Switches context to another Hippo task.

Remarks: NOT IMPLEMENTED

hippo_terminateothertask

Trap: LDA #\$60 : STA \$D640 : CLV

Service: Terminates another Hippo task.

Remarks: NOT IMPLEMENTED

hippo_toggle_force_4502

Trap:	LDA #\$72 : STA \$D640 : CLV
Service:	Toggles the CPU personality between 45GS02 and 6502.
Outputs:	A If bit 5 is set, the CPU is in the 45GS02 personality. If bit 5 is clear, the CPU is in the 6502 personality.
History:	Available since Hyppo 1.2
Remarks:	The others bits of the A register are undefined. Do not expect them to be zero. In the 6502 personality, none of the new opcodes of the 65C02, 65CE02, 4510 or 45GS02 are available. These are replaced with the original — and 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.
Example:	Enables the 45GS02 personality regardless of the CPU's current personality.

```
C10
; Toggle the CPU personality
LDA #$72 : STA $D640 : CLV : BCC error
; Toggle again if bit 5 of A is clear
AND #X00100000 : BEQ C10
```

hippo_toggle_rom_writeprotect

Trap:	LDA #\$70 : STA \$D640 : CLV
Service:	Toggles the write-protection for \$20000 - \$3FFFF.
Outputs:	A If bit 2 is set, \$20000 - \$3FFFF cannot be written to.
History:	Available since Hypo 1.2
Remarks:	The others bits of the A register are undefined. Do not expect them to be zero. If you simply want to disable or enable the protection, you can use hippo_rom_writeenable and hippo_rom_writeprotect .

hippo_writeintotask

Trap: LDA #\$56 : STA \$D640 : CLV

Service: Writes into the memory of another Hyppo task.

Remarks: **NOT IMPLEMENTED**

SYSTEM PARTITION SERVICES

hippo_configsector_apply

Trap: LDA #\$04 : STA \$D642 : CLV

Service: Applies the system configuration sector currently loaded into memory.

History: Available since Hypo 1.2

hippo_configsector_read

Trap: LDA #\$00 : STA \$D642 : CLV

Service: Reads the system configuration sector into memory.

History: Available since Hypo 1.2

hippo_configsector_write

Trap: LDA #\$02 : STA \$D642 : CLV

Service: Writes the system configuration sector from memory.

History: Available since Hypo 1.2

hippo_dmagic_autoset

Trap: LDA #\$06 : STA \$D642 : CLV

Service: Sets the DMAgic revision based on the loaded ROM.

History: Available since Hypo 1.2

FREEZER SERVICES

hippo_freeze_self

Trap: LDA #\$xx : STA \$D67F : CLV

Service: Launches the freezer.

History: Available since Hyppo 1.2

hippo_get_slot_count

Trap: LDA #\$16 : STA \$D642 : CLV

Service: Gets the number of freeze slots.

History: Available since Hyppo 1.2

hippo_locate_freeze_slot

Trap: LDA #\$10 : STA \$D642 : CLV

Service: Locates the first sector of a freeze slot.

History: Available since Hyppo 1.2

hippo_read_freeze_region_list

Trap: LDA #\$14 : STA \$D642 : CLV

Service: Reads the freeze region list.

History: Available since Hyppo 1.2

hippo_unfreeze_from_slot

Trap: LDA #\$12 : STA \$D642 : CLV

Service: Unfreezes from a freeze slot.

History: Available since Hyppo 1.2

K

APPENDIX

Machine Language Monitor

- **Introduction**
- **The MEGA65 Machine Language Monitor**
- **The Matrix Mode/Serial Monitor**

INTRODUCTION

A machine language monitor is a program for inspecting and experimenting with the internal state of a computer's CPU and memory, useful for debugging machine language programs and exploring how the computer works. A typical monitor can view and modify the contents the CPU registers and memory, disassemble machine code, assemble instructions directly to memory, and manipulate memory in other ways, among other features.

The MEGA65 has two separate machine language monitors built in: one in the ROM (the MEGA65 monitor), and one that is part of the Matrix Mode debug interface (the Matrix Mode/serial monitor). Each monitor has different features and uses.

The monitor in the MEGA65 ROM is similar to one that was included with the original C65 ROM, more complete and with more features. It is included in both the MEGA65 closed ROM and the enhanced MEGA65 OpenROM. If you are using an original C65 ROM, its monitor works similarly.

THE MEGA65 MACHINE LANGUAGE MONITOR

The machine language monitor included with the MEGA65 ROM can be invoked in several ways. From BASIC, you can invoke the monitor with the **MONITOR** command.

You can also cause the MEGA65 to boot directly into the monitor by holding the **STOP** key during start-up.

The MEGA65 monitor activates automatically when program execution encounters the **BRK** instruction (byte value 00). You can cause your program to pause and open the monitor at a specific point by including a **BRK** instruction in the program code at that point.

When the monitor starts, it displays the register contents, then waits for a command:

```
MONITOR
PC  SR  AC  XR  YR  ZR  BP  SP
; 000000 00 00 00 00 00 00 F8
```

Numbers

The monitor uses hexadecimal (base 16) to display numbers, such as memory addresses and values. Hexadecimal values are printed without a leading '\$' character, such as 5A or D020.

You can enter a value in hexadecimal, decimal (base 10), octal (base 8), or binary (base 2). Each number format uses a character prefix, as shown in the table below. If you omit the prefix, the monitor assumes the value is in hexadecimal. (This is unlike BASIC, which assumes decimal without a prefix.) You can also enter a byte value as a PETSCII character, with the appropriate prefix.

Hexadecimal is the default number format for assembly instructions as well. This can be confusing! Consider what happens when you tell the monitor to assemble the following instruction:

```
LDA #10
```

A typical assembler interprets this as loading the decimal value 10 (hex 0A) into the accumulator. The monitor assumes this is the hexadecimal value \$10 (decimal 16). To avoid confusion, always provide an explicit prefix character for numbers in assembly instructions. The monitor's disassembler shows all values as hexadecimal starting with the \$ prefix.

All numbers must be literal values. The monitor does not understand arithmetic expressions.

You can use the monitor to show all the base conversions of a given number or PETSCII character. To do this, enter the number as if it were a command, using its prefix. The \$ is required for hexadecimal numbers in this case to distinguish it from other monitor commands.

The following table shows the prefixes for each way a value can be entered:

base	name	prefix	digits	characters	example
16	hexadecimal		0123456789ABCDEF		100
16	hexadecimal	\$	0123456789ABCDEF		\$100
10	decimal	+	0123456789		+256
8	octal	&	01234567		&400
2	binary	%	01		%1000000000
	character	'	all		'A

The Assembler

The monitor has a builtin mini-assembler that can be used to write machine language code using the standard mnemonics like **LDA** or **STA**.

This is not a full assembler like you might use to write large programs. In particular, it does not support symbols, such as labels for subroutines or kernel addresses. Instead of **JSR CHROUT**, you must look up the address of **CHROUT** and enter the number value: **JSR FFD2**

For convenience, you can provide a full address to a branch instruction. The monitor calculates the relative address offset to assemble the instruction.

The assembler knows all of the instructions and address modes of the MEGA65 CPU 45GS02. An instruction like **LDA [TXTPTR],Z** will be assembled as loading the accumulator using a 32-bit pointer at the addresses **TXTPTR**, **TXTPTR+1**, **TXTPTR+2**, **TXTPTR+3**.

Differences from the C65 Monitor

The MEGA65 monitor is similar to the monitor included with the C65. It includes multiple enhancements beyond the C65 monitor, both to make the monitor more generally useful and to extend it to support features specific to the MEGA65 architecture.

The MEGA65 monitor has the following exclusive features:

Addresses:

All addresses are used as 32-bit (4 bytes) addresses. This allows access to the whole MEGA65 address range, which needs 28-bit. This is especially useful for the access to the 8MB RAM blocks called attic RAM at \$8000000 (builtin) and cellar RAM at \$8800000 (optional). Setting bit 31 of an address to 1 gives access to a special (banked) configuration. In this case the I/O area at \$D000 and the ROM area \$6000 - \$7FFF (monitor ROM) and \$E000 - \$FFFF (kernal ROM) overlay the current bank.

Commands:

The additional command **B** displays character bitmaps.

Disk access:

The disk command character knows two more functions: **U1** for reading a sequence of disk blocks to memory and **U2** for writing a memory range to disk blocks. This enables disk disk editing, for example modifying directory entries or can be even used to backup whole floppy contents or disk images. The attic RAM is large enough to hold the contents of 8 complete 1581 floppies.

Disassembler:

The disassembler can decode all additional address modes, like the 32-bit indirect mode $[$nn], Z$ and the compound instructions involving the use of the 32-bit Q register.

Register:

The register displays the full 16-bit stack pointer and the base page register and accepts new settings for them.

Table of MEGA65 Monitor Commands

C	mnemonic	description
A	ASSEMBLE	Assemble a line of 45GS02 code
B	BITMAPS	Display 8x8 bitmaps (characters)
C	COMPARE	Compare two sections of memory
D	DISASSEMBLE	Disassemble a line of 45GS02 code
F	FILL	Fill a section of memory with a value
G	GO	Start execution at specified address
H	HUNT	Find specified data in a section of memory
J	JUMP	Jump to a subroutine at specified address
L	LOAD	Load a file from disk
M	MEMORY	Dump a section of memory
R	REGISTERS	Display the contents of the 45GS02 registers
S	SAVE	Save a section of memory to a disk file
T	TRANSFER	Transfer memory to another location
V	VERIFY	Compare a section of memory with a disk file
X	EXIT	Exit Monitor mode
.	<period>	Assembles a line of 45GS02 code
>	<greater>	Modifies memory
;	<:semicolon>	Modifies register contents
@	<at sign>	Disk command, directory or status
\$	<hex>	Display hex, decimal, octal, and binary value
+	<decimal>	Display hex, decimal, octal, and binary value
&	<octal>	Display hex, decimal, octal, and binary value
%	<binary>	Display hex, decimal, octal, and binary value

A : ASSEMBLE

Format: A address mnemonic operand

Usage: The mini assembler allows entry of machine language instructions using easy to remember mnemonics instead of opcodes. The operand may be entered as hex, decimal, binary or character. Branch targets are automatically converted to relative distances. After each entered instruction, the mini assembler generates the 1-3 byte long machine code, prints this code along with the instruction and advances the program counter. A new line is generated with the command **A** and the new value of the program counter printed. This eases the fast entry of instructions. The assembly input mode is stopped by pressing RETURN only. Any line of the entered code or a line in disassembly format can be changed by moving the cursor into that line and changing the desired element, for example the mnemonic or the operand. Listed hex values before the mnemonic are ignored.

If the monitor shall be reentered after executing the code, the last instruction must be a **BRK** instruction and the program must be called with I/O and monitor ROM active. This is done by setting the bit 31 of the execution address. If the program was entered in bank 0 on address 1500, it should be started with: **G 80001500**.

If the entered code is a subroutine, it must end with a **RTS** instruction.

Remarks: The assembler recognises all 45GS02 instructions of the MEGA65, except the instructions, that use the Q register. These instructions can be entered by typing the NEG NEG prefix explicit. E.g. instead of LDQ \$1234, entering the 3 instructions (on 3 different rows) NEG NEG LDA \$1234 is assembled to the equivalent code.

```
MEGA65 - Xemu [100% 3%] running 3.5MHz

READY.
MONITOR

BS ROM1024 COMMANDS: ABCD CHMTRUWC, > /? /% /LSU
PC SR AC XR YR ZR BP SP NVEBDIZC
; 00CFAF 00 00 00 00 00 00 00 01F8 -----
M 1500 1500
>1500 00 48 45 4C 4C 4F 20 57 4F 52 4C 44 0D 00 00 00 HELLO WORLD...
A 1510 A0 00 LDY #$00
A 1512 B9 00 15 LDA $1500,Y
A 1515 F0 06 BEQ $151D
A 1517 20 D2 FF JSR $FFD2
A 151A C8 INY
A 151B 80 F5 BRA $1512
A 151D 00 BRK
A 151E
G 80001510
HELLO WORLD

BREAK
BS ROM1024 COMMANDS: ABCD CHMTRUWC, > /? /% /LSU
PC SR AC XR YR ZR BP SP NVEBDIZC
; 03151E 02 00 00 00 00 00 00 01F5 -----1-
```

B : BITMAPS

Format: B start

Usage: Displays memory values interpreted as character bitmaps.

Remarks: The B command displays the contents of memory cells bitwise by printing an asterisk for 1 and a dot for 0. The special arrangement of character data with 8 bytes forming one character cell, is considered. 8 characters are displayed for each call.

There are three ROM character sets builtin in the 92XXXX ROMs:

FONT A : REM \$029000 : ASCII [£]†† {()} included
FONT B : REM \$03D000 : serif version of A
FONT C : REM \$02D000 : original C64 font

```

BS MONITOR COMMANDS ABCDE FGHJKLMNPRUXC >P25+8% LSU
PC SR AC XR VR ZR BP SP NVEBDIZC
: 00FFAAZ 00 00 00 00 00 00 01F8 -----
B 2D000
02D000 . ****. . . **. . . ****. . . ***. . . *****. . . *****. . . ***.
02D001 . **. **. . . ***. . . **. **. . . **. . . **. . . **. . . **. . .
02D002 . **. ***. . . **. **. . . ***. . . **. . . **. . . **. . . **. . .
02D003 . **. ***. . . *****. . . **. . . **. . . ***. . . ***. . . ***. . .
02D004 . . . **. . . **. . . **. . . **. . . **. . . **. . . **. . . **. . .
02D005 . **. * . . **. . . **. . . **. . . **. . . **. . . **. . . **. . .
02D006 . ***. . . **. . . ***. . . ***. . . ***. . . ***. . . ***. . .
02D007 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
B 3D000
03D000 . *****. . . **. . . *****. . . ***. . . *****. . . *****. . . *****. . .
03D001 . **. . . ***. . . **. . . **. . . **. . . **. . . **. . . **. . . **. . .
03D002 . **. ***. . . **. . . **. . . **. . . **. . . **. . . **. . . **. . .
03D003 . **. ***. . . **. . . ***. . . **. . . **. . . ***. . . ***. . .
03D004 . **. ***. . . *****. . . **. . . **. . . **. . . **. . . **. . . ***. . .
03D005 . **. . . **. . . **. . . **. . . **. . . **. . . **. . . **. . . **. . .
03D006 . ***. . . **. . . ***. . . ***. . . ***. . . ***. . . ***. . . ***. . .
03D007 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

```

C : COMPARE

Format: **C start end start-b**

Usage: Compares two regions of memory, then lists the addresses whose values differ.

Remarks: The end address is one beyond the last address to compare.

The second memory region to compare starts at **start-b** and is the same size as the region described by **start** and **end**.

Addresses are printed as hex values.

D : DISASSEMBLE

Format: **D [from [to]]**

Usage: Prints a machine language listing for the specified address range assuming, that it contains code. If only one argument is present, the dis-

assembler disassembles the next 21 bytes. If no argument is given, the disassembly continues with the last used disassemble address. The contents are printed as hex values.

Remarks: The rows start with the dot character '..'. This enables direct full screen editing of the disassembly. Typing return in any row will assemble the changed command of the cursor row back to memory, if writable RAM is there. See monitor command ..

The disassembler knows the instruction set of the C65 CPU GS6502. Enhanced instructions from the 45GS02 CPU of the MEGA65 are not recognised.

Example: Using D

```
DL"*
SEARCHING FOR 0:*
LOADING
READY
MONITOR

MONITOR
PC SR AC XR YR ZR SP
: 000000 00 00 00 00 00 F8
D 31F7
: 0031F7 A4 29 LDY $29
: 0031F9 AB 29 00 LDZ $0029
: 0031FC B1 AC LDA ($AC),Y
: 0031FE 91 57 STA ($57),Y
: 003200 EA NOP
: 003201 B2 AE LDA ($AE),Z
: 003203 EA NOP
: 003204 92 59 STA ($59),Z
: 003206 3B DEZ
: 003207 88 DEY
: 003208 10 F2 BPL $31FC
: 00320A 18 CLC
: 00320B A5 AC LDA $AC
```

F : FILL

Format: F start end value

Usage: Fills a region of memory with a value.

Remarks: The end address is one beyond the last address to fill.

G : GO

Format: **G [addr]**

Usage: Executes code at an address. If **addr** is omitted, the current program counter is used as the address.

Remarks: This is equivalent to a **JMP** instruction.

The **G** command without an argument is a convenient way to resume execution of a program after starting the monitor with a **BRK** instruction.

Because this uses a **JMP**, it must not be used to call subroutines. See **J**.

H : HUNT

Format: **H start end byte [byte...]**

Usage: Searches for every occurrence of a byte sequence in a range of memory, then lists the starting address of each occurrence.

Remarks: The end address is one beyond the last address to search.

Addresses are printed as hex values.

J : JUMP

Format: **J addr**

Usage: Jumps to a subroutine at an address.

Remarks: This is equivalent to a **JSR** instruction. When the subroutine returns with RTS, the monitor resumes.

To simply continue execution at an address without expecting to return to the monitor, use **G**.

L : LOAD

Format: **L "filename",unit [start]**

Usage: Loads data from a file into memory.

Remarks: The filename must be surrounded by double-quotes.

The **unit** number (such as 8 or 9) is required.

If the **start** address is omitted, the starting address is taken from the first two bytes of the file. The file must be of type PRG. If **start** is provided, the two byte PRG header is ignored.

M : MEMORY

Format: **M [from [,to]]**

Usage: Prints a memory dump for the given address range. The dump displays memory contents, organised in rows of 16 consecutive addresses starting with the address, given as 1st. argument. The dump continues until a row has been printed, containing the value of the address given as 2nd. argument. If no 2nd. argument is present, the dump displays a full page of 256 bytes in 16 rows. The contents are printed as 16 byte values in hex, followed by the character representation.

Remarks: The rows start with the character '>'. This enables direct full screen editing of the dump. Typing return in any row will write the changed values of the cursor row back to memory, if writable RAM is there. See monitor command >.

Example: Using **M**

```
M 0322A0
>0322A0 20 20 42 41 53 49 43 20 31 30 2E 30 20 20 20 20: BASIC 10.0
>0322B0 56 30 2E 39 2E 39 31 30 31 31 20 20 20 20 20 41:V0.9.910111 A
>0322C0 4C 4C 20 52 49 47 48 54 53 20 52 45 53 45 52 56:LL RIGHTS RESERV
>0322D0 45 44 00 00 60 A2 17 BD E1 22 90 FA 02 CA 10 F7:ED .-.J"J"J"J"
>0322E0 60 ED 78 8A 2D 11 B0 4D 2E EC 2E 8C 23 6C 33 C9:W.-.M...H.3,
>0322F0 2B A8 4C 9C 23 F0 33 C4 2C E3 3D A0 00 20 3B 23+:L.#r3- = .;
>032300 C9 20 F0 F5 C9 3A B0 06 38 E9 30 38 E9 D0 60 A9:R.87087
>032310 50 80 3A A9 3F 80 36 A9 52 80 32 A9 5C 80 2E A9:P.32.67R.274...
>032320 66 80 2A A9 61 80 16 A9 70 80 12 A9 70 80 1E A9:.*A..A..A..A..
>032330 50 80 1A A9 61 80 16 A9 24 80 02 A9 3D DB DA AA:P..A..S..=+
>032340 AB 84 00 20 74 FF FA FB 29 FF 60 A9 24 08 DB DA:U..I..(n.)m-$.+#
>032350 AA AB 85 00 B5 01 C9 20 B0 03 AB 84 00 20 74 FF:U..I..(n.)m-$.+#
>032360 FA FB 28 29 FF 60 08 DB 48 AB 85 00 B5 01 C9 20:U..I..(n.)m-$.+#
>032370 B0 03 AB 84 00 68 20 77 FF FB 28 60 A2 3D 08 DB:U..I..(n.)m-$.+#
>032380 AB 84 00 20 77 FF FB 28 60 6C 04 03 FC 3D 00 20:U..I..(n.)m-$.+#
>032390 FB 22 80 03 20 F9 22 90 FB 6C 0C 03 93 8E 00 C9:U..I..(n.)m-$....
```

R : REGISTERS

Format: R

Usage: Displays the contents of the registers from just before the monitor was started.

Remarks: The output begins with a ; character so that you can change the register values by moving the cursor up to the line, changing the values, then pressing .

S : SAVE

Format: S "filename",unit start end

Usage: Saves a region of memory to a file on disk, of type PRG.

Remarks: The filename must be surrounded by double-quotes.

The **unit** number (such as 8 or 9) is required.

The end address is one beyond the last address to save.

To overwrite an existing file, put 0: before the filename, such as:
"0:MYFILE"

T : TRANSFER

Format: T start end destination-start

Usage: Transfers (copies) a region of memory to another region.

Remarks: The end address is one beyond the last address to copy.

The destination region is assumed to be the same length as the source region. All bytes are copied.

V : VERIFY

Format: V "filename",unit start

Usage: Verifies that a region of memory matches a file on disk.

Remarks: The filename must be surrounded by double-quotes.
The **unit** number (such as 8 or 9) is required.
The length of the data to verify is assumed to be the length of the file.

X : EXIT

Format: X
Usage: Exits the monitor to BASIC.
Remarks: If the monitor was started by a **BRK** instruction in a program, exiting the monitor effectively aborts the program. To continue a paused program, use **G** instead.

. : ASSEMBLE

Format: . address opcode operand
Usage: An alias for **A**.
Remarks: The output of the disassembler starts each line with a . character. You can edit the assembly instructions printed by the disassembler by moving the cursor up to a line, editing it, and pressing . This changes the . to a A so you can see which lines have been reassembled.

> : MODIFY MEMORY

Format: > addr byte [byte...]
Usage: Modifies one or more bytes starting at an address.
Remarks: The output of the **M** command starts each line with a > character. You can edit the memory values printed by the **M** command by moving the cursor up to a line, editing it, and pressing .

; : MODIFY REGISTERS

Format: ; pc sr ac xr yr zr bp sp nvebdizc

- Usage:** Modifies the CPU registers. Arguments are in the format output by the **R** command.
- Remarks:** The output of the **R** command starts the line with a ; character. You can edit the memory values printed by the **R** command by moving the cursor up to the line, editing it, and pressing .

@ : DISK COMMAND

Format: @ [unit] [command]

Usage: Perform a disk command.

Remarks: Without a command, this displays the drive status.

The default unit is 8.

The command can be any Commodore DOS command. For example, to rename a file: @R0:NEWNAME=OLDNAME

The command can be U1 to load raw disk sectors to memory, or U2 to save memory to raw disk sectors. These commands take this form:

@8,U1 mem-start track sector [sector-count]

Tracks are numbered 1-80, and sectors are numbered 0-39. If **sector-count** is omitted, only one sector is read from or written to.

THE MATRIX MODE/SERIAL MONITOR

The Matrix Mode monitor can do something that the MEGA65 monitor cannot: instead of pausing a program to run the monitor, the Matrix Mode monitor runs concurrently with the active program. That is, you can view and modify the memory the MEGA65, *while a program is running*.

This works using dedicated hardware in the MEGA65 design that implements a little helper processor that runs this monitor interface. This hardware has a special access mechanism to the memory and CPU of the MEGA65.

In comparison with the ROM-based monitors that execute on the MEGA65's primary processor, the Matrix Mode monitor has several advantages and disadvantages:

- It can be used while a program is running.
- It can be used, even if the ROM area is being used for program code or data, instead of containing a standard C65 or MEGA65 ROM.

- It can be accessed via the serial debug interface, via the JB1 connector.
- It can be instructed to stop the processor as soon as the program counter (PC) register of the main processor reaches a user specified address. That is, it supports a (single) hardware breakpoint.
- It can be instructed to stop the processor whenever a specified memory address is written to. That is, it supports a “write watch” on a single memory address. The memory address is specified as a full 28-bit address, allowing it to detect memory writes via any means. Note that DMA operations will complete, before the watch point takes effect.
- It can be instructed to stop the processor whenever specific CPU flags are set or cleared, which can also be used to support debugging of programs.
- On some models of the MEGA65, the integrated ROM of the monitor processor is very small, which means that functionality may be limited. This is why, for example, there is no “assemble” command for this monitor. This may be corrected in future core updates for MEGA65 models that have capacity for a larger monitor processor ROM.

Table of Matrix Mode Monitor Commands

C	mnemonic	description
#	HYPERTRAP	Enable/disable CPU hypervisor traps
+	UARTDIVISOR	Set UART bitrate divisor
?	HELP	help
@	CPUMEM	(show memory from CPU context)
B	BREAKPOINT	Set/clear CPU execution break point
D	DISASSEMBLE	Disassemble memory
E	FLAGWATCH	Set/clear CPU flags watch point
F	FILL	Fill memory with a value
G	SETPC	Set CPU program counter
H	HELP	help
I	INTERRUPTS	Enable/disable CPU interrupts
J	DEBUGMON	Various debug functions for the monitor itself
L	LOADMEMORY	Load data into memory
M	MEMORY	Show memory contents
R	REGISTERS	show registers
S	SETMEMORY	Set memory contents
T	TRACE	set CPU trace/run mode
W	MEMORYWATCH	Set/clear memory write watch point
Z	CPUHISTORY	CPU history

Calling the Monitor

To enter or exit the monitor hold down  and press . You will see an animation of green characters raining down from the top of the screen, and then be presented with a simple text terminal interface which is transparent, so that you can see the screen output of your running program at the same time.

: Hypervisor trap enable/disable

Format: # Enable or disable Hypervisor Traps

Usage: #[0|1]

Remarks: If the argument is 1, then Hypervisor traps are enabled, otherwise they are disabled.

It is not confirmed if this command is currently functional.

+ : Set Serial Interface UART Divisor

Format: + Set Serial Interface UART Divisor

Usage: + divisor

Remarks: Sets the divisor for the serial monitor interface. This allows changing the baud rate of the serial monitor interface from the default 2,000,000 bits per second. The baud rate will be equal to $40,500,000 \div (\text{divisor} - 1)$. This affects only the serial UART interface, and does not affect accessing this monitor via the Matrix Mode composited display.

For example, to slow the serial monitor interface down to 19,200 bits per second, the divisor would need to be $40,500,000 \div (19,200 - 1) = 2108$. The + command then requires that you convert this value to hexadecimal, thus the command would be +83c.

Note that this command does *no* sanity checking of the provided value. If you accidentally provide an incorrect value for your needs, you can recover from this situation by activating the Matrix Mode interface by holding down  and tapping , and entering the appropriate command to correct the divisor, e.g., +14 to return to the default of 2,000,000 bits per second.

You must then exit the Matrix Mode again by repeating the  +  key combination, before the serial UART interface will become active again. This is because the Matrix Mode disables the serial UART interface when active.

@ : CPUMEMORY

Format: @ [address]

Usage: Prints a memory dump for the given 16-bit address, as interpreted by the current CPU memory mapping. If you wish to inspect the contents of memory anywhere in the 28-bit address space, use the M command instead.

The dump displays memory contents, organised in rows of 16 consecutive addresses starting with the address. The dump displays a full page of

256 bytes in 16 rows. The contents are printed as 16 byte values in hex, followed by the character representation.

Remarks: If no address is provided, it will show the next 256 bytes.

? or H : HELP

Format: **? or h**

Usage: Displays a (very) brief message identifying the monitor. On some models of the MEGA65 that have more memory available to the monitor processor, this command may display information about each of the available commands.

B : BREAKPOINT

Format: **b [address]**

Usage: Sets or clears the hardware breakpoint. If no address is provided, then the breakpoint will be disabled. Otherwise the breakpoint is set to the provided 16-bit address.

Whenever the program counter (PC) register of the MEGA65's processor equals the value provided to this command, the processor will halt, and the Matrix Mode monitor interface will display the last instruction executed and current register values to alert the user to this event. It does not activate the Matrix Mode display when this occurs. It is normally expected that Matrix Mode will either already be active, or that the user is interacting via the serial interface.

D : DISASSEMBLE

Format: **<d|D> [address]**

Usage: Disassembles and displays the instruction stored at the indicated 28-bit address.

To disassemble instructions from the CPU's current memory context, taking into account current memory banking, prefix the address with 777, e.g., d777080D would disassemble the instruction at \$080D, as currently visible to the MEGA65's processor.

Use D instead of d to disassemble 16 instructions at a time, instead of just one.

E : FLAGWATCH

Format: **e [value]**

Usage: Sets or clears the CPU flag watch point: If no argument is provided, the flag watch point is disabled. If a value is provided, it is assumed to be a 16-bit value, where the first two hexadecimal digits indicate the processor flags that will trigger the watch point if they are set. The second two hexadecimal digits indicate which processor flags will trigger the watch point if they are clear. In this way any combination of processor flag values can be monitored.

This command does not function correctly at the time of writing.

Example: To cause the watch point to trigger when the Negative Flag is asserted, the command e8000 would be used.

F : FILL

Format: **f [start] [end+1] [value]**

Usage: Fills the indicated 28-bit address range with the indicated value.

Remarks: The end address should be one more than the last address that is desired to be filled.

G : SETPC

Format: **g address**

Usage: Sets the Program Counter (PC) register of the MEGA65's processor to the supplied 16-bit address. If the processor is running at the time, execution will immediately proceed from that address. If the processor is halted at the time, e.g., due to the use of the t1 command, the processor remains halted, but with the Program Counter set to the indicated address, ready for when the processor is again allowed to run.

I : INTERRUPTS

Format: `i[0|1]`

Usage: Enables or disables interrupts on the MEGA65's processor. Disabling interrupts can be helpful when single-stepping through a program, as otherwise you will tend to end up only stepping through the interrupt handler code, because the interrupts will happen more frequently than the steps through the code.

Remarks: *This command is known to have problems, and may not currently function.*

J : DEBUGMON

Format: `j [value]`

Usage: Display, and optionally set, internal signals of the matrix mode monitor interface.

L : LOADMEMORY

Format: `l <start addr> <end addr + 1>`

Usage: Fast-load a block of memory via the serial monitor interface. Immediately after sending this command, the bytes of memory to be loaded should be sent to the serial monitor interface. The bytes are read as-is, and thus should be provided as natural bytes, not encoded in hexadecimal. This allows loading data at approximately 200KB per second at the default serial baud rate of 2,000,000 bits per second.

M : MEMORY

Format: `<m|M> [address]`

Usage: Prints a memory dump for the given 28-bit address. If you wish to inspect the contents of memory as currently seen by the processor's current banking configuration, use the `m` command instead.

The dump displays memory contents, organised in rows of 16 consecutive addresses starting with the address. The dump displays a full page of 256 bytes in 16 rows. The contents are printed as 16 byte values in hex, followed by the character representation.

Remarks: If no address is provided, it will show the next 256 bytes.

R : REGISTERS

Format: r

Usage: Displays the current value of various processor registers and flags, as well as a disassembly of the most recently executed instruction.

S : SETMEMORY

Format: s addr <value ...>

Usage: Sets the contents of the indicated memory location to the supplied value. If more than one space-separated value is provided, then multiple consecutive memory locations will be set.

This command uses 28-bit addresses, and therefore ignores the current selected memory banking configuration.

T : TRACE

Format: t<0|1|c>

Usage: Selects the trace or run mode of the processor: t0 means that the processor runs freely, t1 halts the processor, and tc runs the processor in continuous-trace mode, where it displays each instruction and the register values immediately following its execution, as though t1 had been selected, and the user were to then immediately press return or enter to request the next instruction to be executed.

If t1 is selected, pressing enter or return in the Matrix Mode monitor will cause the next instruction to be executed.

The t0 command is also used following the triggering of a break-point or watch-point, to allow the processor to resume.

W : WATCHPOINT

Format: w [address]

Usage: Sets or clears the hardware watch-point. If no address is provided, then the watch-point will be disabled. Otherwise the watch-point is set to the provided 28-bit address.

Whenever the MEGA65's processor writes to the address provided to this command, the processor will halt, and the Matrix Mode monitor interface will display the last instruction executed and current register values to alert the user to this event. It does not activate the Matrix Mode display when this occurs. It is normally expected that Matrix Mode will either already be active, or that the user is interacting via the serial interface.

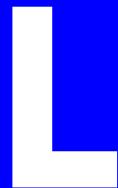
Z : CPUHISTORY

Format: **z [address]**

Usage: Displays information about the instructions recently executed by the MEGA65's processor.

Remarks: *This command is suspected to not be correctly operational at the time of writing.*

APPENDIX



F018-Compatible Direct Memory Access (DMA) Controller

- F018A/B DMA Jobs
- MEGA65 Enhanced DMA Jobs
- Texture Scaling and Line Drawing
- Inline DMA Lists
- Audio DMA
- 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 40MB or copying 20MB per second. For example, it is possible to copy a complete 8KB 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 256MB 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 Amiga™ series 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.

F018A/B DMA JOBS

To execute a DMA job using the F018 series of DMA controllers, you must construct the list of DMA jobs in memory, and then write the address of this list into the DMA address registers. The DMA job will execute when you write to the ADDRLSBTRIG register (\$D700). For this reason you must write the MSB and bank number of the DMA list int0

\$D701 and \$D702 first, and the LSB only after having set these other two registers. If you wish to execute multiple DMA jobs using the same list structure in memory, you can simply write to ADDRLSBTRIG again after updating the list contents – provided that no other program has modified the contents of \$D701 or \$D702. Note that BASIC 65 uses the DMA controller to scroll the screen, so it is usually safest to always write to all three registers.

When ADDRLSBTRIG has been written to, the DMA job completes immediately. Unlike on the C65, the DMA controller is part of the processor of the MEGA65. This means that the processor stops trying to execute instructions until the DMA job has completed. The only exception to this, is that Audio DMA continues, and will steal cycles from any other DMA activity, to ensure that audio playback is not affected.

This behaviour means that unlike on the C65, DMA jobs cannot be interrupted. If your program has sensitive timing requirements, you may need to break larger DMA jobs into several smaller jobs. This is somewhat mitigated by the high speed of the MEGA65's DMA, which is able to fill memory at 40.5MB per second and copy memory at 20.25MB per second, compared with circa 3.5MB and 1.7MB per second on a C65. This allows larger DMA jobs to be executed, without needing to worry about the impact on real-time elements of a program. For example, it is possible to fill an 80 column 50 row text screen using the MEGA65's DMA controller in just 200 microseconds.

F018 DMA Job List Format

The MEGA65's DMA controller supports the two different DMA job list formats used by the original F018 part that was used in the earlier C65 prototypes (upto Revision 2B) and the F018B and later revisions used in the Revision 3 – 5 C65 prototypes. The main difference is the addition of a second command byte, as the following tables show:

It is important to know which style the DMA controller is expecting. The MEGA65's Hypervisor sets the mode based on the detected version of C65 ROM, if one is running. If it is an older one, then the F018 style is expected, otherwise the newer F018B style is expected. You can check which style has been selected by querying bit 0 of \$D703: If it is a 1, then the newer F018B 12 byte list format is expected. If it is a 0, then the older F018 11 byte list format is expected. The expected style can be set by writing to this register.

Unless you are writing software that must also run on a C65 prototype, you should most probably use the MEGA65's Enhanced DMA Jobs, where the list format is explicitly specified in the list itself. As the Enhanced DMA Jobs are an extension of the F018/F018B DMA jobs, you should still read the following, unless you are already familiar with the behaviour of the F018 DMA controller.

F018 11 byte DMA List Structure

Offset	Contents
\$00	Command LSB
\$01	Count LSB
\$02	Count MSB
\$03	Source Address LSB
\$04	Source Address MSB
\$05	Source Address BANK and FLAGS
\$06	Destination Address LSB
\$07	Destination Address MSB
\$08	Destination Address BANK and FLAGS
\$09	Modulo LSB
\$0a	Modulo MSB

* The Command MSB is \$00 when using this list format.

F018B 12 byte DMA List Structure

Offset	Contents
\$00	Command LSB
\$01	Count LSB
\$02	Count MSB
\$03	Source Address LSB
\$04	Source Address MSB
\$05	Source Address BANK and FLAGS
\$06	Destination Address LSB
\$07	Destination Address MSB
\$08	Destination Address BANK and FLAGS
\$09	Command MSB
\$0a	Modulo LSB / Mode
\$0b	Modulo MSB / Mode

The structure of the command word is as follows:

Bit(s)	Contents
0 - 1	DMA Operation Type
2	Chain (i.e., another DMA list follows)
3	Yield to interrupts
4	MINTERM -SA,-DA bit
5	MINTERM -SA,DA bit
6	MINTERM SA,-DA bit
7	MINTERM SA,DA bit
8 - 9	Addressing mode of source
10 - 11	Addressing mode of destination
12 - 15	RESERVED. Always set to 0's

The command field take the following four values:

Value	Contents
%00 (0)	Copy
%01 (1)	Mix (via MINTERMs)
%10 (2)	Swap
%11 (3)	Fill

* Only Copy and Fill are implemented at the time of writing.

The addressing mode fields take the following four values:

Value	Contents
%00 (0)	Linear (normal) addressing
%01 (1)	Modulo (rectangular) addressing
%10 (2)	Hold (constant address)
%11 (3)	XY MOD (bitmap rectangular) addressing

* Only Linear, Modulo and Hold are implemented at the time of writing.

The BANK and FLAGS field for the source address allow selection of addresses within a 1MB address space. To access memory beyond the first 1MB, it is necessary to use an Enhanced DMA Job with the appropriate option bytes to select the source and/or destination MB of memory. The BANK and FLAGS field has the following structure:

Bit(s)	Contents
0 - 3	Memory BANK within the selected MB
4	HOLD, i.e., do not change the address
5	MODULO, i.e., apply the MODULO field to wrap-around within a limited memory space
6	DIRECTION. If set, then the address is decremented instead of incremented.
7	I/O. If set, then I/O registers are visible during the DMA controller at \$D000 - \$DFFF.

Performing Simple DMA Operations

For information on using the DMA controller from BASIC 65, refer to the **DMA** BASIC command in Chapter/Appendix B on page B-77.

To use the DMA controller from assembly language, set up a data structure with the DMA list, and then set \$D702 - \$D700 to the address of the list. For example, to clear the screen in C65-mode by filling it with spaces, the following routine could be used:

```
LDA #$00      ; DMA list exists in BANK 0
STA $D702
LDA #>dmalist ; Set MSB of DMA list address
STA $D701
LDA #<dmalist ; Set LSB of DMA list address, and execute DMA
STA $D700
RTS

dmalist:
.byte $03    ; Command low byte: FILL
.word 2000   ; Count: 80x25 = 2000 bytes
.word $0020  ; Fill with value $20
.byte $00    ; Source bank (ignored with FILL operation)
.word $0800  ; Destination address where screen lives
.byte $00    ; Screen is in bank 0
.byte $00    ; Command high byte
.word $0000  ; Modulo (ignored due to selected command)
```

It is also possible to execute more than one DMA job at the same time, by setting the CHAIN bit in the low byte of the command word. For example to clear the screen as above, and also clear the colour RAM for the screen, you could use something like:

```

LDA #$00      ; DMA list exists in BANK 0
STA $D702
LDA #>dmalist ; Set MSB of DMA list address
STA $D701
LDA #<dmalist ; Set LSB of DMA list address, and execute DMA
STA $D700
RTS

dmalist:
.byte $07    ; Command low byte: FILL + CHAIN
.word 2000   ; Count: 80x25 = 2000 bytes
.word $0020   ; Fill with value $20
.byte $00    ; Source bank (ignored with FILL operation)
.word $0800   ; Destination address where screen lives
.byte $00    ; Screen is in bank 0
.byte $00    ; Command high byte
.word $0000   ; Modulo (ignored due to selected command)

; Second DMA job immediately follows the first
.byte $03    ; Command low byte: FILL
.word 2000   ; Count: 80x25 = 2000 bytes
.word $0001   ; Fill with value $01 = white
.byte $00    ; Source bank (ignored with FILL operation)
.word $F800   ; Destination address where colour RAM lives
.byte $01    ; colour RAM is in bank 1 ($1F800-$1FFF)
.byte $00    ; Command high byte
.word $0000   ; Modulo (ignored due to selected command)

```

Copying memory is very similar to filling memory, except that the command low byte must be modified, and the source address field must be correctly initialised. For example, to copy the character set from where it lives in the ROM at \$2D000 - \$2DFFF to \$5000, you could use something like:

```

LDA #$00      ; DMA list exists in BANK 0
STA $D702
LDA #>dmalist ; Set MSB of DMA list address
STA $D701
LDA #<dmalist ; Set LSB of DMA list address, and execute DMA
STA $D700
RTS

dmalist:
.byte $00    ; Command low byte: COPY
.word $1000  ; Count: 4KB = 4096
.word $0000  ; Copy from $x0000
.byte $02    ; Source bank = $02 for $2xxxx
.word $5000  ; Destination address where screen lives
.byte $00    ; Screen is in bank 0
.byte $00    ; Command high byte
.word $0000  ; Modulo (ignored due to selected command)

```

It is also possible to perform a DMA operation from BASIC 2 in C64 mode by POKEing the necessary values, after first making sure that MEGA65 or C65 I/O mode has been selected by writing the appropriate values to \$D02F (53295). For example, to clear the screen in C64 BASIC 2 using the DMA controller, you could use something like:

```

10 rem enable mega65 I/O
20 poke53295,asc("g"):poke53295,asc("s")
30 rem dma list in data statements
40 data 3: rem command lsb = fill
50 data 232,3 : rem screen is 1000 bytes = 3*256+232
60 data 32,0: rem fill with space = 32
70 data 0: rem source bank (unused for fill)
80 data 0,4: rem screen address = 1024 = 4*256
90 data 0: rem screen lives in bank 0
100 data 0: rem command high byte
110 data 0,0: rem modulo (unused in this job)
120 rem put dma list at $c000 = 49152
130 fori=0to11:read:a:poke49152+i,a:next
140 rem execute job
150 poke55042,0: rem dma list is in bank 0
160 poke55041,192: rem dma list is in $c0xx
170 poke55040,0: rem dma list is in $xx00, and execute

```

While this is rather cumbersome to do each time, if you wanted to clear the screen again, all you would need to do would be to **POKE 55040, 0** again, assuming that the DMA list and DMA controller registers had not been modified since the previous time the DMA job had been run.

The HOLD, I/O and other options can also be used to create interesting effects. For example, to write a new value to the screen background colour very quickly, you could copy a region of memory to \$D021, with the I/O flag set to make the I/O register visible for writing in the DMA job, and the HOLD flag set, so that the same address gets written to repeatedly. This will write to the background colour at a rate of 20.5MHz, which is almost as fast as the video pixel clock (27MHz). Thus we can change the colour almost every pixel.

With a little care, we can make this routine such that it takes exactly one raster-line to run, and thus draw vertical raster bars, or to create a kind of frankenstein video mode that uses a linear memory layout – at the cost of consuming all of the processor's time during the active part of the display.

The following example does this to draw vertical raster bars on the screen. This program assumes that the MEGA65 is set to PAL. For NTSC, the size of the DMA transfer would need to be decreased a little. The other thing to note with this program, is that it uses MEGA65 Enhanced DMA Job option \$81 to set the destination megabyte in memory to \$FFxxxx, and the bank is set to \$D, and the destination address to \$0021, to form the complete address \$FFD0021. This is the true location of the VIC-IV's border colour register. The program is written using ACME-compatible syntax.

```
basicheader:  
    tti;; 2020 SYS 2061  
    tt!word $80a,2020  
    tt!byte $9e,$32,$30,$36,$31,0,0,0  
  
    tti;; Actual code begining at $080d = 2061  
Main:  
    tt!sei  
    tt!lda #$47      tti; enable MEGA65 I/O  
    tt!sta $D02f  
    tt!lda #$53  
    tt!sta $d02f  
    tt!lda #65 tti; Set CPU speed to fast  
    tt!sta 0  
    tt!lda #0      tti; disable screen to show only the border  
    tt!sta $d011  
  
    tt!lda $d012      tti; Wait until start of the next raster  
raster+sync:ttt tti; before beginning loop for horizontal alignment  
    tt!cmp $d012  
    tt!beq raster+sync  
  
    tti;; The following loop takes exactly one raster line at 40.5MHz in PAL  
loop:  
    tt!jsr triggereddma  
    tt!jmp loop  
  
triggerdma:  
    tt!lda #0ttt tti; make sure F018 list format  
    tt!sta $d703  
  
    tt!lda #0      tti; dma list bank  
    tt!sta $d702  
    tt!lda #>rasterdmalist  
    tt!sta $d701  
    tt!lda #<rasterdmalist  
    tt!sta $d705  
    tt!rts
```

```
rasterdmaList:
    !byte $81,$ff,$00
    !byte $00 !I!I; COPY
    !word 619 !I!I; DMA transfer is 619 bytes long
    !word rastercolours!I!I; source address
    !byte $00 ; source bank
    !word $0020 !I!I; destination address
    !byte $1d ; destination bank + HOLD
    !I;; unused modulo field
    !word $0000
```

```
rastercolours:
    !byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    !byte 0,0,0,11,11,11,12,12,12,15,15,15,1,1,1,15,15,15,12,12,12,11,11,11,0,0,0
    !byte 0,0,0,6,6,6,4,4,4,14,14,14,3,3,3,1,1,1,3,3,3,14,14,14,14,4,4,6,6,6,0,0,0
    !byte 0,0,0,11,11,11,12,12,12,15,15,15,1,1,1,15,15,15,12,12,12,11,11,11,0,0,0
    !byte 0,0,0,6,6,6,4,4,4,14,14,14,3,3,3,1,1,1,3,3,3,14,14,14,14,4,4,6,6,6,0,0,0
```

MEGA65 ENHANCED DMA JOBS

The MEGA65's implementation of the DMAagic 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 that 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.

These options are also used to achieve advanced features, such as hardware texture scaling at up to 20Mpixels per second, and hardware line drawing at up to 40Mpixels per second. These advanced functions are implemented by allowing complex calculations to be made to the source and/or destination address of DMA jobs as they execute.

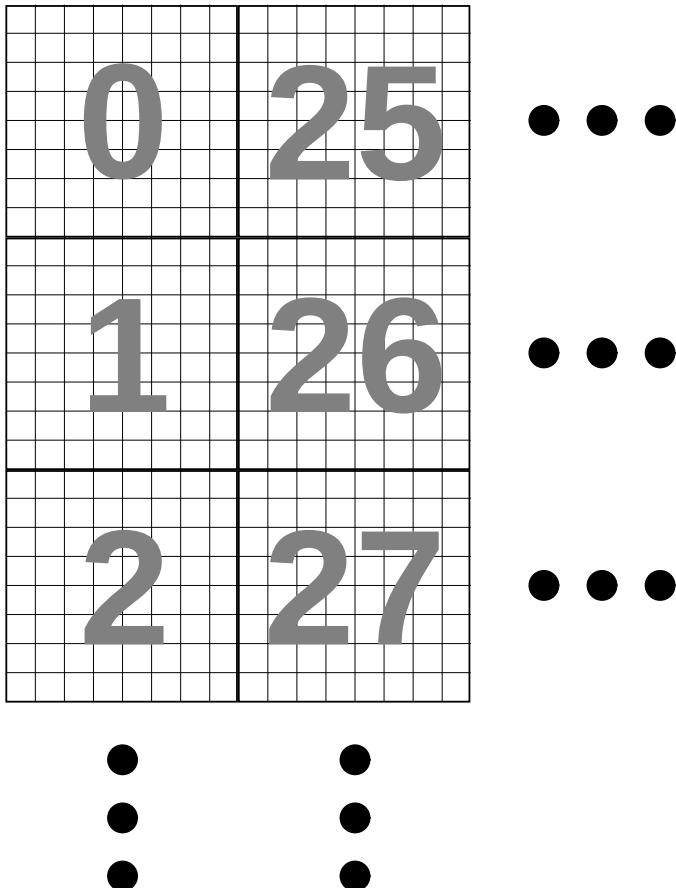
The list of valid job option tokens is:

\$00	End of job option list
\$06	Disable use of transparent value
\$07	Enable use of transparent value
\$0A	Use 11 byte F011A DMA list format
\$0B	Use 12 byte F011B DMA list format
\$53	Enable 'Shallan Spiral' Mode
\$80	Source address bits 20 - 27
\$81	Destination address bits 20 - 27
\$82	Source skip rate (256 ^{ths} of bytes)
\$83	Source skip rate (whole bytes)
\$84	Destination skip rate (256 ^{ths} of bytes)
\$85	Destination skip rate (whole bytes)
\$86	Transparent value (bytes with matching value are not written)
\$87	Set X column bytes (LSB) for line drawing destination address
\$88	Set X column bytes (MSB) for line drawing destination address
\$89	Set Y row bytes (LSB) for line drawing destination address
\$8A	Set Y row bytes (MSB) for line drawing destination address
\$8B	Slope (LSB) for line drawing destination address
\$8C	Slope (MSB) for line drawing destination address
\$8D	Slope accumulator initial fraction (LSB) for line drawing destination address
\$8E	Slope accumulator initial fraction (MSB) for line drawing destination address
\$8F	Line Drawing Mode enable and options for destination address (set in argument byte): Bit 7 = enable line mode, Bit 6 = select X or Y direction, Bit 5 = slope is negative.
\$97	Set X column bytes (LSB) for line drawing source address
\$98	Set X column bytes (MSB) for line drawing source address
\$99	Set Y row bytes (LSB) for line drawing source address
\$9A	Set Y row bytes (MSB) for line drawing source address
\$9B	Slope (LSB) for line drawing source address
\$9C	Slope (MSB) for line drawing source address
\$9D	Slope accumulator initial fraction (LSB) for line drawing source address
\$9E	Slope accumulator initial fraction (MSB) for line drawing source address
\$9F	Line Drawing Mode enable and options for source address (set in argument byte): Bit 7 = enable line mode, Bit 6 = select X or Y direction, Bit 5 = slope is negative.

TEXTURE SCALING AND LINE DRAWING

The DMAgic supports an advanced internal address calculator that allows it to draw scaled textures and draw lines with arbitrary slopes on VIC-IV FCM video displays.

For texture scaling, the FCM screen must be arranged vertically, as shown below:



By lining the characters into vertical columns like this, advancing vertically by one pixel adds a constant 8 bytes each time, as shown below:



The source and destination skip rates also allow setting the scaling factors. A skip rate of \$0100 this corresponds to stepping \$01.00 pixels. To use the vertically stacked FCM layout as the target for copying vertical lines of textures, then the destination skip rate should be \$0800, i.e., 8.0 bytes per pixel. This would copy a vertical line of texture data without scaling. By setting the source stepping to < \$0100 will cause some pixels to be repeated, effectively zooming the texture in, while setting the source stepping to > \$0100 will cause some pixels to be skipped, effectively zooming the texture out. The destination stepping does not ordinary need to be adjusted. Note that the texture data must be stored with each vertical stripe stored contiguously, so that this mode can be used.

For line drawing, the DMA controller needs to know the screen layout, specifically, what number must be added to the address of a rightmost pixel in one column of FCM characters in order to calculate the address of the pixel appearing immediately to its

right. Similarly, it must also know how much must be added to the address of a bottom most pixel in one row of FCM characters in order to calculate the address of the pixel appearing immediately below it. This allows for flexible screen layout options, and arbitrary screen sizes. You must then also specify the slope of the line, and whether the line has the X or Y as its major axis, and whether the slope is positive or negative.

The file `test_290.c` in the <https://github.com/mega65/mega65-tools> repository provides an example of using these facilities to implement hardware accelerated line drawing. This is very fast, as it draws lines at the full DMA fill speed, i.e., approximately 40,500,000 pixels per second.

INLINE DMA LISTS

Normally you have to setup a separate area of memory that contains the DMA list, and then load the address of that area into the DMA address registers at \$D70x. Because the MEGA65's DMA controller is part of the CPU, it supports an additional mode that is very convenient, called inline DMA list mode.

This mode works like Enhanced DMA mode, except that the DMA list is read starting from the current value of the Program Counter (PC) register. To use this mode, write any value to \$D707, to immediately trigger a DMA job, with the list in the bytes immediately following the instruction that writes to \$D707.

The DMA list can be a single job, or chained, as with any other DMA job. The real magic is that the Program Counter gets set to the next address after the end of the DMA list, and that the DMA list is read from the CPU's current memory mapping. This means that you can execute code with DMA lists from any bank of memory, without having to worry about which bank it is in.

For example, the following code would clear the C65-mode screen, before flashing the border endlessly:

```

STA $D707
.byte $00 ; end of job options
.byte $03 ; fill
.word 2000 ; count
.word $0020 ; value
.byte $00 ; src bank
.word $0800 ; dst
.byte $00 ; dst bank
.byte $00 ; cmd hi
.word $0000 ; modulo / ignored

foo:
INC $d820
JMP foo

```

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 value 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 \$D729. 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-hand 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 nibble 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.

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

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D703	55043					-			EN018B
D704	55044					ADDRMB			
D705	55045					ETRIG			
D706	55046					ETRIGMAPD			
D70E	55054					ADDRLSB			
D711	55057	AUDEN	BLKD	AUD-WRBLK	NOMIX	-		AUDBLKTO	
D71C	55068				CH0RVOL				
D71D	55069				CH1RVOL				
D71E	55070				CH2LVOL				
D71F	55071				CH3LVOL				
D720	55072	CH0EN	CH0LOOP	CH0SGN	CH0SINE	CH0STP	-		CH0SBITS
D721	55073				CH0BADDR				
D722	55074				CH0BADDRC				
D723	55075				CH0BADDRM				
D724	55076				CH0FREQL				
D725	55077				CH0FREQC				
D726	55078				CH0FREQM				

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D727	55079				CH0TADDRL				
D728	55080				CH0TADDRM				
D729	55081				CH0VOLUME				
D72A	55082				CH0CURADDRL				
D72B	55083				CH0CURADDRC				
D72C	55084				CH0CURADDRM				
D72D	55085				CH0TMRADDRL				
D72E	55086				CH0TMRADDRC				
D72F	55087				CH0TMRADDRM				
D730	55088	CH1EN	CH1LOOP	CH1SGN	CH1SINE	CH1STP	-		CH1SBITS
D731	55089				CH1BADDRL				
D732	55090				CH1BADDRC				
D733	55091				CH1BADDRM				
D734	55092				CH1FREQL				
D735	55093				CH1FREQC				
D736	55094				CH1FREQM				
D737	55095				CH1TADDRL				
D738	55096				CH1TADDRM				
D739	55097				CH1VOLUME				
D73A	55098				CH1CURADDRL				
D73B	55099				CH1CURADDRC				
D73C	55100				CH1CURADDRM				
D73D	55101				CH1TMRADDRL				
D73E	55102				CH1TMRADDRC				
D73F	55103				CH1TMRADDRM				
D740	55104	CH2EN	CH2LOOP	CH2SGN	CH2SINE	CH2STP	-		CH2SBITS
D741	55105				CH2BADDRL				
D742	55106				CH2BADDRC				
D743	55107				CH2BADDRM				
D744	55108				CH2FREQL				
D745	55109				CH2FREQC				
D746	55110				CH2FREQM				
D747	55111				CH2TADDRL				
D748	55112				CH2TADDRM				
D749	55113				CH2VOLUME				
D74A	55114				CH2CURADDRL				
D74B	55115				CH2CURADDRC				
D74C	55116				CH2CURADDRM				

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D74D	55117					CH2TMRADDRL			
D74E	55118					CH2TMRADDRC			
D74F	55119					CH2TMRADDRM			
D750	55120	CH3EN	CH3LOOP	CH3SGN	CH3SINE	CH3STP	-		CH3SBITS
D751	55121					CH3BADDRL			
D752	55122					CH3BADDRC			
D753	55123					CH3BADDRM			
D754	55124					CH3FREQL			
D755	55125					CH3FREQC			
D756	55126					CH3FREQM			
D757	55127					CH3TADDRL			
D758	55128					CH3TADDRM			
D759	55129					CH3VOLUME			
D75A	55130					CH3CURADDRL			
D75B	55131					CH3CURADDRC			
D75C	55132					CH3CURADDRM			
D75D	55133					CH3TMRADDRL			
D75E	55134					CH3TMRADDRC			
D75F	55135					CH3TMRADDRM			

- **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
- **AUDBLKTO** Audio DMA block timeout (read only) DEBUG
- **AUDEN** Enable Audio DMA
- **AUDWRBLK** Audio DMA block writes (samples still get read)
- **BLKD** Audio DMA blocked (read only) DEBUG
- **CHORVOL** Audio DMA channel 0 right channel volume
- **CH1RVOL** Audio DMA channel 1 right channel volume
- **CH2LVOL** Audio DMA channel 2 left channel volume
- **CH3LVOL** Audio DMA channel 3 left channel volume
- **CHXBADDRC** Audio DMA channel X base address middle byte
- **CHXBADDRL** Audio DMA channel X base address LSB

- **CHXBADDRM** Audio DMA channel X base address MSB
- **CHXCURADDRC** Audio DMA channel X current address middle byte
- **CHXCURADDRL** Audio DMA channel X current address LSB
- **CHXCURADDRM** Audio DMA channel X current address MSB
- **CHXEN** Enable Audio DMA channel X
- **CHXFREQC** Audio DMA channel X frequency middle byte
- **CHXFREQL** Audio DMA channel X frequency LSB
- **CHXFREQM** Audio DMA channel X frequency MSB
- **CHXLOOP** Enable Audio DMA channel X looping
- **CHXSBITS** Audio DMA channel X sample bits (11=16, 10=8, 01=upper nybl, 00=lower nybl)
- **CHXSGN** Enable Audio DMA channel X signed samples
- **CHXSINE** Audio DMA channel X play 32-sample sine wave instead of DMA data
- **CHXSTP** Audio DMA channel X stop flag
- **CHXTADDRL** Audio DMA channel X top address LSB
- **CHXTADDRM** Audio DMA channel X top address MSB
- **CHXTMRADDRC** Audio DMA channel X timing counter middle byte
- **CHXTMRADDRL** Audio DMA channel X timing counter LSB
- **CHXTMRADDRM** Audio DMA channel X timing counter MSB
- **CHXVOLUME** Audio DMA channel X playback volume
- **ENO18B** DMA enable F018B mode (adds sub-command byte)
- **ETRIG** Set low-order byte of DMA list address, and trigger Enhanced DMA job, with list address specified as 28-bit flat address (uses DMA option list)
- **ETRIGMAPD** Set low-order byte of DMA list address, and trigger Enhanced DMA job, with list in current CPU memory map (uses DMA option list)
- **NOMIX** Audio DMA bypasses audio mixer

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.

M

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-III).
- Support for **32KB of 8-bit Colour/Attribute RAM** (up from 2KB on the VIC-III), to support very large screens.
- **HDTV $720 \times 576 / 800 \times 600$ native resolution** at both 50Hz and 60Hz for **PAL and NTSC**, with **VGA and digital video** output.
- **81MHz pixel clock** (up from ~ 8 MHz with the VIC-II/III), which enables a wide range of new features.
- New 16-colour (16×8 pixels per character cell) and 256-colour (8×8 pixels per character cell) **full-colour text modes**.
- Support for up to **8,192 unique characters in a character set**.
- **Four 256-colour palette banks** (versus the VIC-III's single palette bank), each supporting **23-bit colour depth** (versus the VIC-III's 12-bit colour depth), and which can be rapidly alternated to create even more colourful graphics than is possible with the VIC-III.
- Screen, bitmap, colour and character data can be positioned at any **address with byte-level granularity** (compared with fixed 1KB - 16KB boundaries with the VIC-II/III)
- **Virtual screen dimensioning**, which combined with byte-level data position granularity provides effective **hardware support for scrolling and panning in both X and Y directions**.
- **New sprite modes**: Bitplane modification, **full-colour** (15 foreground colours + transparency) and tiled modes, allowing a wide variety of new and exciting sprite-based effects
- The ability to stack sprites in a bit-planar manner to produce **sprites with up to 256 colours**.

- Sprites can use 64 bits of data per raster line, allowing **sprites to be 64 pixels wide** when using VIC-II/III mono/multi-colour mode, or 16 pixels wide when using the new VIC-IV full-colour sprite mode.
- **Sprite tile mode**, which allows a sprite to be repeated horizontally across an entire raster line, allowing sprites to be used to create animated backgrounds in a memory-efficient manner.
- Sprites can be configured to use a **separate 256-colour palette** to that used to draw other text and graphics, allowing for a more colourful display.
- **Super-extended attribute mode** which uses two screen RAM bytes and two colour RAM bytes per character mode, which supports a wide variety of new features including **alpha-blending/anti-aliasing, hardware kerning/variable-width characters**, hardware horizontal/vertical flipping, alternate palette selection and other powerful features that make it easy to create highly dynamic and colourful displays.
- **Raster-Rewrite Buffer** which allows **hardware-generated pseudo-sprites**, similar to “bobs” on Amiga™ computers, but with the advantage that they are rendered in the display pipeline, and thus do not need to be un-drawn and re-drawn 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:

First Value Hex (Decimal)	Second Value Hex (Decimal)	Effect	VIC-IV Specific?
\$00 (0)	\$00 (0)	Only VIC-II registers visible (all VIC-III and VIC-IV new registers are hidden)	No
\$A5 (165)	\$96 (150)	VIC-III new registers visible	No
\$47 (71)	\$53 (83)	Both VIC-III and VIC-IV new registers visible	Yes
\$45 (69)	\$54 (84)	No VIC-II/III/IV registers visible. 45E100 Ethernet controller buffers are visible instead	Yes

Detecting VIC-II/III/IV

Detecting which generation of the VIC-II/III/IV a machine is fitted with can be important for programs that support only particular generations, or that wish to vary their graphical display based on the capabilities of the machine. While there are many possibilities for this, the following is a simple and effective method. It relies on the fact that the VIC-III and VIC-IV do not repeat the VIC-II registers throughout the I/O 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:

```

8 REM IN C65-MODE WE CANNOT SAFELY WRITE TO $D02F, SO WE TEST A DIFFERENT WAY
10 IF PEEK($D018) AND 32 THEN GOTO 65
20 POKE $D000,1:POKE $D02F,71:POKE $D02F,83
30 POKE $D000+256,0:IF PEEK($D000)=1 THEN PRINT"VIC-IV PRESENT":END
40 POKE $D000,1:POKE $D02F,165:POKE $D02F,150
50 POKE $D000+256,0:IF PEEK($D000)=1 THEN PRINT"VIC-III PRESENT":END
60 PRINT "VIC-II PRESENT":END
65 REM WE ASSUME WE HAVE A C65 HERE
70 V1=PEEK($D050):V2=PEEK($D050):V3=PEEK($D050)
80 IF V1<>V2 OR V1<>V3 OR V2<>V3 THEN PRINT "VIC-IV PRESENT":END
90 GOTO 40

```

Line 10 of this program checks whether the screen is a multiple of 2KB. As the screen on the C64 is located at 1KB, 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 I/O enabled will be one of the red palette registers. After writing to 53248 + 256, the program 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 720×576 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 1280×200 and 1280×400 display modes of the VIC-III are not currently supported, as they cannot be meaningfully displayed on any modern monitor, and no software is known to support or use this feature.

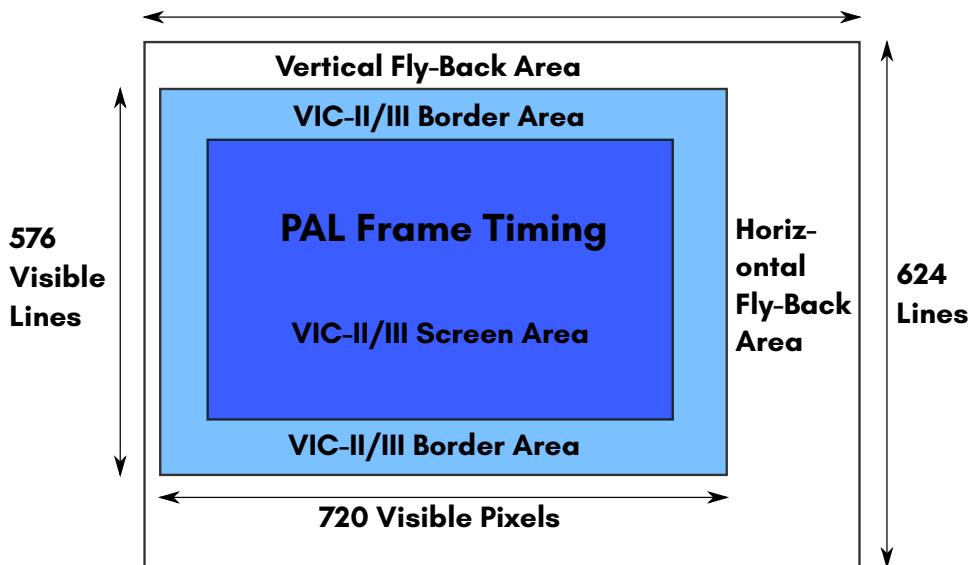
Frame Timing

Frame timing is designed to match that of the 6502 + VIC-II combination of the C64. Both PAL and NTSC timing is supported, and the number of cycles per logical raster line, the number of raster lines per frame, and the number of cycles per frame are all adjusted accordingly. To achieve this, the VIC-IV ordinarily uses HDTV 576p 50Hz (PAL) and 480p 60Hz (NTSC) video modes, with timing tweaked to be as close as possible to double-scan PAL and NTSC composite TV modes as used by the VIC-II.

The VIC-IV produces timing impulses at approximately 1MHz which are used by the 45GS02 processor, so that the correct effective frequency is provided when operating at the 1MHz, 2MHz and 3.5MHz C64, C128 and C65 compatibility modes. This allows the single machine to switch between accurate PAL and NTSC CPU timing, as well as video modes. The exact frequency varies between PAL and NTSC modes, to mimic the behaviour of PAL versus NTSC C64, C128 and C65 processor and video timing.

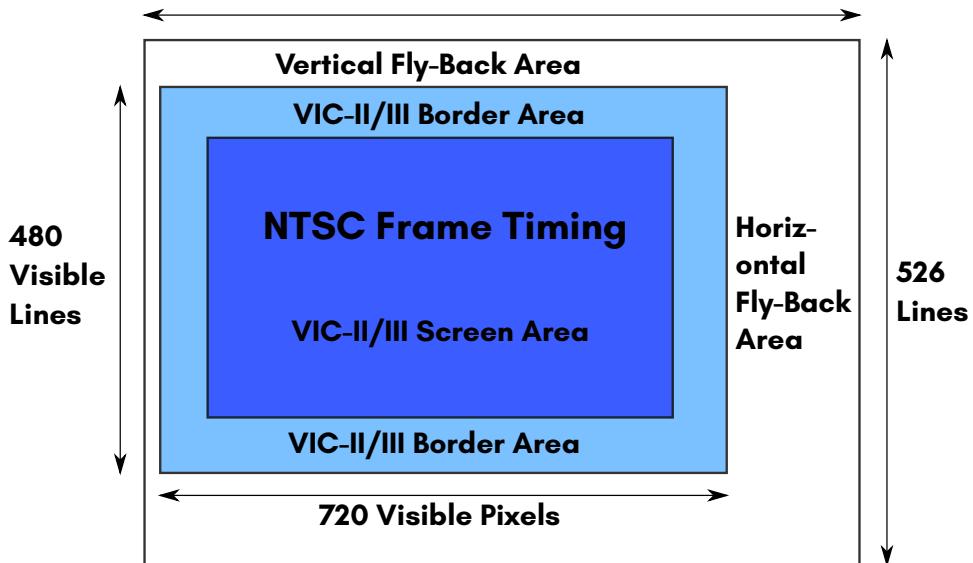
The PAL frame is constructed from 624 physical raster lines, consisting of 864 pixel clock ticks. The pixel clock is 27MHz, which is 1/3 the VIC-IV pixel clock. The visible frame is 720×576 pixels, the entirety of which can be used in VIC-IV mode. In VIC-II and VIC-III modes, the border area reduces the usable size to 640×400 pixels. In VIC-II mode and VIC-III 200H modes, the display is double scanned, with two 31.5 micro-second physical rasters corresponding to a single 63 micro-second VIC-II-style raster line. Thus each frame consists of 312 VIC-II raster lines of 63 micro-seconds each, exactly matching that of a PAL C64.

**864 Horizontal Ticks
(31.5 µSec per line)**



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 720×480 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 640×400 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 µSec per line)**



As these HDTV video modes are not supported by all VGA monitors, a compatibility mode is included that provides a 640×480 VGA-style mode. However, as the pixel clock of the MEGA65 is fixed at 27MHz, this mode runs at 63Hz. Nonetheless, this should work on the vast majority of VGA monitors. There should be no problem with the PAL / NTSC modes when using the digital video output of the MEGA65 with the vast majority of 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 REM ENABLE C65+MEGA65 I/O
20 IF PEEK($D018)<32 THEN POKE $D02F,ASC("6"):POKE $D02F,ASC("$")
30 REM CHECK NTSC BIT
40 NTSC=PEEK($D06F) AND 128
50 REM DISPLAY STATE AND ASK FOR TOGGLE
60 PRINT"MEGA65 IS IN ";:IF NTSC THEN PRINT"NTSC MODE":ELSE PRINT"PAL MODE"
70 INPUT"SWITCH MODES (Y/N)? ",A$
80 REM TOGGLE NTSC BIT
90 IF A$="Y" THEN POKE $D06F,PEEK($D06F) XOR 128:ELSE END
100 REM DISPLAY NEW STATE
110 NTSC=PEEK($D06F) AND 128
120 PRINT"MEGA65 IS IN ";:IF NTSC THEN PRINT"NTSC MODE":ELSE PRINT"PAL MODE"

```

Physical and Logical Rasters

Physical rasters per frame refers to the number of actual raster lines in the PAL or NTSC Enhanced Definition TV (EDTV) video modes used by the MEGA65. Logical Rasters refers to the number of VIC-II-style rasters per frame. Each logical raster consists of two physical rasters per line, since EDTV modes are double-scan modes compared with the original PAL and NTSC Standard Definition TV modes used by the C64. The frame parameters of the VIC-IV for PAL and NTSC are as follows:

Standard	Cycles per Raster	Physical Rasters per Frame	Logical Rasters per Frame
PAL	63	626	312
NTSC	65	526	263

The result is that the frames on the VIC-IV consist of exactly the same number of ~1MHz CPU cycles as on the VIC-II.

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 16MB of direct access RAM for video data, however at present, all existing models provide only 384KB of addressable RAM. In MEGA65 systems, the second block of 128KB of RAM (spanning from 128KB-256KB in the memory map) is typically used to hold a C65-compatible ROM, leaving 256KB of RAM available to the user. If software is written to avoid the need to use C65 ROM routines, then the entire 384KB of RAM can be used by the program.

All MEGA65 models presently support 32KB of colour RAM, however there are plans for the latest R3 board to support 64KB of colour RAM (or possibly even 128KB).

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:

```
REM ENABLE C65+MEGA65 I/O
IF PEEK($D018)<32 THEN POKE $D02F,ASC("G"):POKE $D02F,ASC("$")
POKE $D060,$45:POKE $D061,$23:POKE $D062,$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 command WPOKE can be used to write two bytes as a word into a memory or I/O location. Therefore, we use WPOKE to write \$00 into \$D068 and \$12 into \$D069, and an additional POKE to write the high byte \$A into \$D06A by dividing the address by 65536:

```
REM ENABLE CG5+MEGA65 I/O
IF PEEK($D018)<32 THEN POKE $D02F,ASC("G"):POKE $D02F,ASC("S")
REM HEX $41200 IS EASILY DIVIDED IN ITS 3 BYTES $00, $12, $4
REM WPOKE SETS THE LOWER TWO BYTES IN ONE COMMAND AND
REM THE FOLLOWING POKE SETS THE UPPER BYTE
A=$41200
WPOKE $D068,A
POKE $D06A,A/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:

```
REM ENABLE CG5+MEGA65 I/O
IF PEEK($D018)<32 THEN POKE $D02F,ASC("G"):POKE $D02F,ASC("S")
REM SET COLPTR TO $4000, SPLITS INTO $00 LSB and $40 MSB
POKE $D064,$00
POKE $D065,$40
```

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:

```
REM ENABLE CG5+MEGA65 I/O
IF PEEK($D018)<32 THEN POKE $D02F,ASC("G"):POKE $D02F,ASC("$")
POKE $D06C,(800-INT(800/65536)*65536) AND 255
POKE $D06D,INT(800/256) AND 255
POKE $D06E,(PEEK($D06E) AND 128)+INT(800/65536)
```

The location of each sprite image remains a multiple of 64 bytes, thus allowing for up to 65,536 unique sprite images to be used at any point in time, if the system is equipped with sufficient RAM (4MB or more). In this mode, the VIC-II 16KB banking is ignored, and the location of sprite data is simply $64 \times$ the pointer value. For example, to have the data for a sprite at \$C000 (49152 decimal), this would be sprite location 768, because $49152 \div 64 = 768$. We then need to split 768 into high and low bytes, to set the two pointer bytes: $768 = 256 \times 3$, with remainder 0, so this would require the two sprite pointer bytes to be 0 (low byte, which comes first) and 3 (high byte). Thus if the sprite pointers were located at \$7F8 (2040 decimal), setting the first sprite to sprite image 768 could be done with something like:

```
POKE 2040,768-256*INT(768/256)
POKE 2041,INT(768/256)
```

HOT REGISTERS

Some VIC-IV registers support features similar to the VIC-II and VIC-III, but with expanded capabilities. For backwards compatibility, writing to specific VIC-II and VIC-III registers also causes related VIC-IV registers to reset with consistent values. If you write to any of these registers, you may need to update the VIC-IV registers after you do so.

For example, the lower four bits of register \$D018 (**CB**) set the VIC-II character set address, as a multiple of 1 KiB. VIC-IV can locate the character set to any 24-bit address using \$D06A (**CHARPTRBNK**), \$D068 (**CHARPTRLSB**), and \$D069 (**CHARPTRMSB**). If you set CB, the VIC-IV registers will also be updated to match.

The complete set of VIC-II registers that affect VIC-IV registers include:

- \$D011 (53265): RB8, ECM, BMM, BLNK, RSEL, YSCL
- \$D016 (53270): RST, MCM, CSEL, XSCL
- \$D018 (53272): VS, CB
- \$D031 (53297): VIC-III modes: H640, FAST, ATTR, BPM, V400, H1280, MONO, INT
- The VIC-II bank bits of \$DD00 (56576) (CIA 2 PORTA)

Whenever any of those registers are modified, even by writing the existing value back into them, various VIC-IV registers will be updated immediately, if the HOTREG bit is set. The registers that are modified during this process are listed below. Note that some of these registers are internal to the VIC-IV, and cannot be directly queried or modified by the user. Where this is the case, no addresses are listed for the registers.

- **X position of the left side border edge.** This internal register is updated set the left side border to the width indicated in the Single Side Border Width registers (\$D05C contains the LSB, and bits 0 – 5 of \$D05D contain the MSB of the side border width. Note that the width of the side border is based on the low-level video frame dimensions, not the display screen size of the video mode. The 38/40 column field of \$D016 is set to 38 columns, the left border edge will appear 14 pixels to the right of its normal position.
- **X position of the right side border edge.** This is the same as the left side border edge, but for the right-hand edge of the screen. Note that if the 38/40 column flag is set to 38 columns, that the right border edge is moved 17 pixels to the left of its normal position.
- **Y position of the top border edge (\$D048 LSB, bits 0 – 3 of \$D049 MSB).** This internal register is set to the normal top position of the screen, minus the

value of the RASLINE0 field in bits 0 – 5 of \$D06F. If the 24/25 rows field of \$D011 is set to 24 rows, then the edge of the top border will be lowered by 8 raster lines.

- **Y position of the bottom border edge (\$D04A LSB, bits 0 – 3 of \$D04B MSB).** This internal register is set to the normal top position of the screen, plus 400 raster lines, to create the normal 400px tall primary display area within the borders. If the 24/25 rows field of \$D011 is set to 24 rows, then the edge of the top border will be raised by 8 raster lines.
- **Character Generator Vertical Scale (\$D05B).** This register is set to 0 for V200 or 1 for V400 modes, to cause each row of pixels in a character to be either 1 or 2 pixels tall, respectively, according to the V400 flag.
- **Number of character rows to display (\$D07B).** This register is set to either 25-1 = 24 or 50-1 = 49 to display either 25 or 50 rows of text. Note that when \$D011 is used to bring the vertical borders inwards to reduce the number of visible character rows, that the VIC-IV still draws all 25 or 50 rows.
- **X Position Where Character Display Starts (\$D04C LSB, bits 0 – 3 of \$D04D MSB).** This register is set to a position relative to the edge of the 40-column wide text display, plus $2 \times$ the smooth scrolling position indicated in \$D016.
- **Y Position Where Character Display Starts (\$D04E LSB, buts 0 – 3 of \$D04F MSB).** This register is set to the top edge of the vertical border, minus the VIC-II First Raster adjustment register (bits 0 – 5 of \$D06F), plus any offset due to the vertical smooth-scroll bits in \$D011.
- **Virtual Row Width (\$D058 LSB, \$D059 MSB), i.e., the number of bytes of screen and colour RAM that the VIC-IV advances when displaying each successive row of characters.** This register is set to 40 if the H640 flag is clear, or to 80 if the H640 flag is set, making the advance match the number of characters to be displayed.
- **Display Row Width (\$D05E LSB, bits 4 – 5 of \$D063 MSB).** This register is set to 40 if the H640 flag is clear, or to 80 if the H640 flag is set.
- **Base Address of Screen RAM (\$D060 – \$D062, representing a 24-bit address).** This address is reset to the address as computed by reference to \$D018 and \$DD00, as on the C64.
- **VIC-II Sprite Pointer Address (\$D06C – \$D06E, representing a 24-bit address).** This register is reset to the normal location at the end of the screen memory of the current mode. If the H640 flag is set, then this will be at the end of 2KB screen RAM area, or if the H640 flag is not set, it will point to the end of the 1KB screen RAM area, as on a C64.

- **Character Set Base Address (\$D068 – \$D06A, representing a 24-bit address).** Note that the hot register function sets only the lower 16 bits of the character set address. That is, \$D06A is not cleared. This is an intentional behaviour, that makes it easier to replace the character set in existing VIC-II-oriented software with another character set in another bank of RAM.
- **Colour RAM Base Address (\$D064 LSB, \$D065 MSB).** These registers are reset to zero, causing the VIC-IV to expect the colour RAM for the screen to be in the first part of the colour RAM, to be compatible with the VIC-II and VIC-III.

This behavior of the VIC-II registers is intended primarily for legacy software. It can be disabled by clearing the HOTREG ("hot register") signal: bit 7 of \$D05D (53341). If you do clear the HOTREG flag, you will then need to update all some or all of these registers when you wish to change the video mode parameters. Alternatively, you can re-enable HOTREG, make a change via that method, and then restore the value of any of the affected registers that you wished to keep after the change.

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 DMAgic to assist, scrolling a 320×200 256-colour display requires 128,000 clock cycles in the best case (reading and writing $320 \times 200 = 64000$ bytes). At 3.5MHz on the C65 this would require about 36 milli-seconds, or about 2 complete video frames. Thus for smooth scrolling of such a display, a double buffered arrangement would be required, which would consume 128,000 of the 131,072 bytes of memory.

In contrast, the well known character modes of the VIC-II are widely used in games, due to their ability to allow a small amount of screen memory to select which 8×8 block of pixels to display, allowing very rapid scrolling, reduced memory consumption, and effective hardware acceleration of animation of common elements. Thus the focus of improvements in the VIC-IV has been on character mode. As bitmap mode on the VIC-II is effectively a special case of character mode, with implied character numbers, it comes along free for the ride on the VIC-IV, and will only be mentioned in the context of a very few bitmap-mode specific improvements that were trivial to make, and it thus seemed foolish to not implement, in case they find use.

Displaying more than 256 unique characters via “Super-Extended Attribute Mode”

The primary innovation is the addition of the Super-Extended Attribute Mode. The VIC-II already uses 12 bits per character: Each 8×8 cell is defined by 12 bits of data: 8 bits of screen RAM data, by default from \$0400 – \$07E7 (1024 – 2023 decimal), indicating which characters to show, and 4 bits of colour data from the 1K nibble colour RAM at \$D800 – \$DBFF (55296 – 56319 decimal). The VIC-III of the C65 uses 16 bits, as the colour RAM is now 8 bits, instead of 4, with the extra 4 bits of colour RAM being used to support attributes (blink, bold, underline and reverse video). It is recommended to revise how this works, before reading the following. A good introduction to the VIC-II text mode can be found in many places. Super-Extended Attribute mode doubles the number of bits per character used from the VIC-III’s 16, to 32: Two bytes of screen RAM and two bytes of colour/attribute RAM.

Super-Extended Attribute Mode is enabled by setting bit 0 in \$D054 (53332 decimal). Remember to first enable VIC-IV mode, to make this register accessible. When this bit is set, two bytes are used for each of the screen memory and colour RAM for each character shown on the display. Thus, in contrast to the 12 bits of information that the C64 uses per character, and the 16 bits that the VIC-III uses, the VIC-IV has 32 bits of information. How those 32 bits are used varies slightly among the particular modes. The default is as follows:

Default Bit Fields (when GOTOX bit is cleared):

Bit(s)	Function when GOTOX bit is cleared
Screen RAM byte 0	
Bits 7 - 0	Lower 8 bits of character number, the same as the VIC-II and VIC-III
Screen RAM byte 1	
Bits 7 - 5	Trim pixels from right-hand side of character (bits 0 - 2)
Bits 4 - 0	Upper 5 bits of character number (bits 8 - 12), allowing addressing of 8,192 unique characters
Colour RAM byte 0	
Bit 7	Vertically flip the character
Bit 6	Horizontally flip the character
Bit 5	Alpha blend mode (leave 0, discussed later)
Bit 4	GOTOX is cleared (set to 0) GOTOX allows repositioning of characters along a raster via the Raster-Rewrite Buffer, (discussed later). Must be set to 0 for displaying characters
Bit 3	If set, Full-Colour characters use 4 bits per pixel and are 16 pixels wide (less any right-hand side trim bits), instead of using 8 bits per pixel. When using 8 bits per pixels, the characters are the normal 8 pixels wide
Bit 2	Trim pixels from right-hand side of character (bit 3)
Bits 1 - 0	Number of pixels to trim from top or bottom of character
Colour RAM byte 1	
If VIC-II multi-colour mode is enabled:	
Bits 7 - 4	Upper 4 bits of colour of character
If VIC-III extended attributes are enabled:	
Bit 7	Hardware underlining of character

continued ...

...continued

Bit(s)	Function when GOTOX bit is <u>cleared</u>
Bit 6	Hardware bold attribute of character *
Bit 5	Hardware reverse video enable of character *
Bit 4	Hardware blink of character
Remaining bit-field is common:	
Bits 3 - 0	Low 4 bits of colour of character

* Enabling BOLD and REVERSE attributes at the same time on the MEGA65 selects an alternate palette, effectively allowing 512 colours on screen, but each 8×8 character can use colours only from one 256 colour palette.

If the GOTOX bit is set, some of the fields have different meanings:

Bit Fields when GOTOX bit is set:

Bit(s)	Function when GOTOX bit is <u>set</u>
Screen RAM byte 0	
Bits 7 - 0	Lower 8 bits of new X position to start drawing the next character, relative to the start of character drawing. Setting to 0 causes the next character to be drawn over the top of the left-most character.
Screen RAM byte 1	
Bits 7 - 5	FCM Character data Y offset: Characters display normally when set to zero. When non-zero, $8 \times$ the value is added to the character address. With careful planning, this can be used to smoothly vertically scroll multiple layers of RRB content.
Bits 4 - 3	RESERVED, set to 0
Bits 1 - 0	Upper 2 bits of new X position
Colour RAM byte 0	
Bit 7	If set, then background/transparent pixels will not be drawn for subsequent characters, allowing layering
Bit 6	If set, the following characters will be rendered as background, allowing sprites to appear in front of them, even when sprites are set to background.
Bit 5	RESERVED, set to 0

continued ...

...continued

Bit(s)	Function when GOTOX bit is <u>set</u>
Bit 4	GOTOX, set to 1 GOTOX allows repositioning of characters along a raster via the Raster-Rewrite Buffer, discussed later). Must be set to 0 for displaying characters
Bit 3	ROWMASK. If set, then the pixel row mask is used to determine which pixel rows of the following characters should be rendered. This can be used to vertically scroll characters using the Raster-Rewrite Buffer, by drawing each character twice, once shifted down on the screen line on which it appears, and a second time, shifted up in the following screen line, and masked so that only the pixel rows belonging to the scrolled character are displayed, and not data from either before or after that character's data.
Bit 2	If set, the following characters will be rendered as foreground, regardless of their colouring, allowing sprites to appear behind them.
Bits 1 - 0	RESERVED, set to 0
Colour RAM byte 1	
Bits 7 - 0	Pixel row mask flags

We can see that we still have the C64 style bottom 8 bits of the character number in the first screen byte. The second byte of screen memory gets five extra bits for that, allowing $2^{13} = 8,192$ different characters to be used on a single screen. That's more than enough for unique characters covering an 80×50 screen (which is possible to create with the VIC-IV). The remaining bits allow for trimming of the character. This allows for variable width characters, which can be used to do things that would not normally be possible, such as using text mode for free horizontal placement of characters (or parts thereof). This was originally added to provide hardware support for proportional width fonts.

For the colour RAM, the second byte (byte 1) is the same as the C65, i.e., the lower half providing four bits of foreground colour, as on the C64, plus the optional VIC-III extended attributes. The C65 specifications document describes the behaviour when more than one of these are used together, most of which are logical, but there are a few combinations that behave differently than one might expect. For example, combining bold with blink causes the character to toggle between bold and normal mode. Bold mode itself is implemented by effectively acting as bit 4 of the foreground colour value, causing the colour to be drawn from different palette entries than usual.

However, if you do not need VIC-III extended attributes, you can instead use the upper four bits of the second byte of colour RAM to contain more bits for the colour index, allowing selection from the full range of 256 colour entries. This mode is activated by enabling the VIC-II's multi-colour mode while full-colour mode is active.

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 384KB of main memory using registers \$D060 - \$D062 (53344 - 53346 decimal). The colour RAM can be located anywhere in the 32KB colour RAM. Only the first 1 or 2KB of the colour RAM is visible at \$D800 - \$DBFF or \$D800 - \$DFFF (if the CRAM2K signal is set in bit 0 of \$D030, 53296 decimal). Thus if using a screen larger than 40×25 characters use of the DMA controller or some other means is required to access the full amount of colour RAM. Therefor we will initially discuss using Super-Extended Attribute Mode with a 40×25 character display.

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 I/O 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, with one character requiring two screen RAM bytes, thus 80 screen RAM bytes are required to display 40 characters. Similarly 80 colour RAM bytes are required as well.

To understand this visually, it is helpful to first consider the normal C64 screen memory layout:

That is, each character cell uses one byte of screen RAM, and the addresses increase smoothly, both within lines, and between lines. Super-Extended Attribute Mode requires two bytes per character cell. So if you set \$D054 to \$05, for example, you will get screen addresses like this:

There are two things to notice in the above table: First, the address advances by two bytes for each character cell, because two bytes are required to define each character. Second, the start address of each screen line still only advances by 40 (\$28 in hexadecimal). This isn't what we really want, because it means that half of the previous row will get displayed again on each current row. This is fixed by setting the number of bytes to advance each screen row in \$D058 (LSB) and \$D059 (MSB). So in this case, we want to increase the number of bytes skipped each line from 40 bytes, to 80 bytes, which we can do by setting \$D058 to 80 (\$50 in hexadecimal), and \$D059 to 0. This gives us a screen layout like this:

It is possible to use Super-Extended Attribute Mode from C65-mode, by setting the screen to 80 columns, as the C65 ROM sets up 2KB for both the screen RAM and colour RAM, and this automatically sets \$D058 and \$D059 to the correct value for $40 \times 2 = 80$ bytes per screen line. The user need only to treat each character pair as a single Super-Extended Attribute character, and to enable Super-Extended Attribute Mode, as described above.

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.

Managing a Super-Extended Attribute Mode screen in this way using BASIC 65 is of course rather a hack, and is only suggested as a relatively simple way to begin experimenting. You will almost certainly want to quickly move to using custom screen handling code, most probably in assembly, to manipulate Super-Extended Attribute Mode screens, although this approach of using BASIC 65 can be quite powerful, by allowing use of existing screen scrolling and other manipulations.

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 (FCM)

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, without sacrificing horizontal resolution.

To do this, each character now requires 64 bytes of data. The address of the data is $64 \times$ the character number, regardless of the character set address. FCM should normally be used with Super-Extended Attribute Mode (SEAM), so that more than 256 unique characters can be address. As SEAM allows the selection of 8,192 unique characters, this allows FCM character data to be placed anywhere in the first 512KB of chip RAM (but note that most models of the MEGA65 have only 384KB of chip RAM).

Please note that the pixel value \$ff will not select the corresponding colour code directly. Instead, it will select the colour code defined by the colour RAM.

Nibble-colour (16 colours per character) Text Mode (NCM)

The Nibble-Colour Mode (NCM) for text is similar to Full-Colour Text Mode, except that each byte of data describes two pixels using 4 bits each. This makes the NCM unique, because the characters will be 16 pixels wide, instead of the usual 8 pixels wide. This can be used to create colourful displays, without using as much memory as FCM, because fewer characters are required to cover the screen. Unlike the VIC-II's MCM, this mode does not result in a loss of horizontal resolution.

In NCM the lower four bits of the pixel colour comes from the upper or lower four bits of the pixel data. The upper four bits of the colour code come from the colour RAM data for the displayed character. This makes it possible to use all palette entries in NCM, although the limitation of 16 colours per character remains. Similar to the behaviour of FCM, the pixel data value \$f will select the pixel colour set in the colour RAM.

A further advantage of NCM is that it uses fewer bus cycles per pixel than FCM, because fewer character data fetches need to occur per raster line. Together with the reduced memory requirements, this makes NCM particularly useful for creating colourful multiple layers of graphics. This allows the VIC-IV to display arcade style displays with more colours than many 16-bit computers.

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 32-bit and 64-bit 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 MEGAWat!? screenshot

XXX Example program

Raster Re-write 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.

If the GOTO bit is set for a character, and the character width reduction bits are also set, they are interpreted as a Y offset to add to the character data address, but only in Full Colour Mode. Setting Y=1 causes the character data to be fetched from 8 bytes later, i.e., the first row of character data will come from the address where the second row of character data would normally be fetched. Similarly for increased values the character data will be fetched from further character rows. With careful arrangement of characters in memory, it is possible to use this feature to provide free vertical placement of soft sprites, without needing to copy the character data.

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.

A good introduction to the Raster Re-write Buffer and its uses can be found in this video:

<https://www.youtube.com/watch?v=00bm5uBeBos&feature=youtu.be>

One important aspect of the RRB, is that the VIC-IV will display only the character data to the left of, and including, the last drawn character. This means that if you use the GOTO token to overwrite multiple layers of graphics, you must either make sure that the last layer reaches to the right-hand edge of the display, or you must include a GOTO token that moves the render position to the right-hand edge of the display.

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 $64 \text{ bits} \div 4 \text{ bits per pixel} = 16 \text{ pixels wide}$.

3. Tiled mode, where the sprite is drawn repeatedly until the end of the raster line. Tiled mode should normally only be used with Extra Wide sprite mode, as the tiling always occurs using the full 64-bit sprite data. Thus if you use tiled mode with normal 24 pixel wide mono or multi-colour sprites, the tiling will treat each 2 and 2/3 rows of sprite data as a single row, resulting in garbled displays.

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). Enabling Full Colour mode for a sprite implicitly enables extended width mode, causes these sprites to be 16 pixels wide.

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:

```
10 PRINT CHR$(147)
20 REM C65/C64-MODE DETECT
30 IF PEEK(53272) AND 32 THEN GOTO 100
40 POKE 53295,ASC("G"):POKE 53295,ASC("S")
100 REM SETUP SPRITE
110 AD=4096 :REM $1000 SPRITE ADDR
120 TC=10 :REM TRANSPARENT COLOUR
130 SPR=PEEK(53356)+PEEK(53357)*256 :REM GET SPRITE TABLE ADDRESS
140 POKE SPR,AD/64 :REM SET SPRITE ADDRESS
150 FOR I=AD TO AD+168 :REM CLEAR SPRITE WITH TC
160 POKE I,TC+TC*16 :REM ONE BYTE = 2 PIXEL
170 NEXT
180 POKE 53287,TC :REM SET TRANSPARENT COLOUR
190 POKE 53248,100 :REM PUT SPRITE...
200 POKE 53249,100 :REM ON SCREEN AT 100,100
210 POKE 53355,1 :REM MAKE SPRITE 0 16-COLOUR
220 POKE 53335,1 :REM MAKE SPRITE 0 USE 16X4-BITS
230 POKE 53269,1 :REM ENABLE SPRITE 0
240 GOSUB 900 :REM READ MULTI-COLOUR SPRITE
250 END

900 REM LOAD SPRITE FROM DATA
910 READ N$:IF N$="END" THEN RETURN
920 GOSUB 1000 :REM DECODE LINE
930 GOTO 910
```

```
1000 REM DECODE STRING OF NIBBLES IN NS AT ADDRESS AD
1010 IF LEN(NS)<>16 THEN PRINT "ILLEGAL SPR DATA!":END
1020 FOR I=1 TO 16 STEP 2
1030 N=(ASC(MID$(NS,I,1))-ASC("0")) :REM HIGH NYB
1040 IF N<0 THEN N=TC :REM . IS TRANSPARENT
1050 M=(ASC(MID$(NS,I+1,1))-ASC("0")) :REM LOW NYB
1060 IF M<0 THEN M=TC :REM . IS TRANSPARENT
1070 POKE AD,(N AND 15)*16 + (M AND 15):REM SET 2 PIXELS
1080 AD=AD+1 :REM ADVANCE AD
1090 NEXT I
1100 RETURN

1998 REM SPRITE DATA
1999 REM . = TRANSPARENT, 0-0 = COLOURS 0 TO 15
2000 DATA "...AAFF...HHCC...""
2010 DATA "...AAFF.....HHCC.."
2020 DATA "AAFF.....HHCC."
2030 DATA "AFF...0EE...HHC."
2040 DATA "FF..0E0GGG0E..HH."
2050 DATA "..0E0GGGGGGGG0E.."
2060 DATA ".0GGGGGGGGGGGG0E.."
2070 DATA ".CGGGGGGGGGGGGG0E.."
2080 DATA "0GGGGGGGGGGGGGG0E.."
2090 DATA "0GGGGGGGGGGGGGG0E.."
2100 DATA "0GGGGGGGGGGGGGGGG0E.."
2110 DATA "0GGGGGGGBGBGGGGGG0E.."
2120 DATA "0GGGGBBBBBBB0GGGGGG0E.."
2130 DATA ".0GGGGBBBBBGGGGGG0E.."
2140 DATA ".CGGGGGGBGGGGGGGG0E.."
2150 DATA "..0GGGGGGGGGGGG0E.."
2160 DATA "II..0E0GGG0E..KK.."
2170 DATA "DII...0EE...KKE.."
2180 DATA "DDII.....KKEE.."
2190 DATA ".DDII.....KKEE.."
2200 DATA "..DDII...KKEE.."
2210 DATA "END"
```

VIC-II / C64 REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D000	53248						S0X		
D001	53249						S0Y		
D002	53250						S1X		
D003	53251						S1Y		
D004	53252						S2X		
D005	53253						S2Y		
D006	53254						S3X		
D007	53255						S3Y		
D008	53256						S4X		
D009	53257						S4Y		
D00A	53258						S5X		
D00B	53259						S5Y		
D00C	53260						S6X		
D00D	53261						S6Y		
D00E	53262						S7X		
D00F	53263						S7Y		
D010	53264						SXMSB		
D011	53265	RC8	ECM	BMM	BLNK	RSEL			YSCL
D012	53266						RC		
D013	53267						LPX		
D014	53268						LPY		
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	MRIRQ
D01B	53275						BSP		
D01C	53276						SCM		
D01D	53277						SEXK		
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				-				C128-FAST

- **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
- **LWX** 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 bits 0 to 7
- **RC8** 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
- **SBC** sprite/foreground collision indicate bits
- **SCM** sprite multicolour enable bits
- **SCREENCOL** screen colour (16 colour)
- **SE** sprite enable bits
- **SEX_X** sprite horizontal expansion enable bits
- **SEXY** sprite vertical expansion enable bits
- **SNX** sprite N horizontal position
- **SNY** sprite N vertical position
- **SPRMCO** Sprite multi-colour 0
- **SPRMC1** Sprite multi-colour 1
- **SPRNCOL** sprite N colour / 16-colour sprite transparency colour (lower nybl)
- **SSC** sprite/sprite collision indicate bits
- **SXMSB** sprite horizontal position MSBs
- **VS** screen address (\times 1KiB)
- **XSCL** horizontal smooth scroll
- **YSCL** 24/25 vertical smooth scroll

VIC-III / C65 REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D020	53280					BORDERCOL			
D021	53281					SCREENCOL			
D022	53282					MC1			
D023	53283					MC2			
D024	53284					MC3			
D025	53285					SPRMC0			
D026	53286					SPRMC1			
D02F	53295					KEY			
D030	53296	ROME	CROM9	ROMC	ROMA	ROM8	PAL	EXTSYNC	CRAM2K
D031	53297	H640	FAST	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

- **BNPIX** Display Address Translator (DAT) Bitplane N port
- **BORDERCOL** display border colour (256 colour)
- **BPCOMP** Complement bitplane flags
- **BPM** Bit-Plane Mode
- **BPX** Bitplane X
- **BPY** Bitplane Y
- **BXADEVN** Bitplane X address, even lines
- **BXADODD** Bitplane X address, odd lines
- **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 (0) or RAM (1) entries for colours 0 - 15
- **PALBLUE** blue palette values (reversed nybl order)
- **PALGREEN** green palette values (reversed nybl order)
- **PALRED** red palette values (reversed nybl order)
- **ROM8** Map C65 ROM \$8000
- **ROMA** Map C65 ROM \$A000
- **ROMC** Map C65 ROM \$C000

- **ROME** Map C65 ROM \$E000
- **SCREENCOL** screen colour (256 colour)
- **SPRMC0** Sprite multi-colour 0 (8-bit for selection of any palette colour)
- **SPRMC1** Sprite multi-colour 1 (8-bit for selection of any palette colour)
- **V400** Enable 400 vertical pixels
- **VPOS** Bitplane Y Offset

VIC-IV / MEGA65 SPECIFIC REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D020	53280				BORDERCOL				
D021	53281				SCREENCOL				
D022	53282				MC1				
D023	53283				MC2				
D024	53284				MC3				
D025	53285				SPRMC0				
D026	53286				SPRMC1				
D02F	53295				KEY				
D048	53320				TBDRPOS				
D049	53321			SPRBPMEN			TBDRPOS		
D04A	53322				BBDRPOS				
D04B	53323			SPRBPMEN			BBDRPOS		
D04C	53324				TEXTXPOS				
D04D	53325			SPRTILEN			TEXTXPOS		
D04E	53326				TEXTYPOS				
D04F	53327			SPRTILEN			TEXTYPOS		
D050	53328				XPOSLSB				
D051	53329	NORRDEL	DBLRR			XPOSMSB			
D052	53330				FNRASTERLSB				
D053	53331	FNRST	SHDEMU		-		FNRASTERMSB		
D054	53332	ALPHEN	VFAST	PALEMU	SPRH640	SMTH	FCLRHI	FCLRLO	CHR16
D055	53333				SPRHGTEN				
D056	53334				SPRHGHT				
D057	53335				SPRX64EN				

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D058	53336					LINESTEPLSB			
D059	53337					LINESTEPMSB			
D05A	53338					CHRXSCL			
D05B	53339					CHRYSCL			
D05C	53340					SDBDRWDLSB			
D05D	53341	HOTREG	RSTDELEN			SDBDRWDMSB			
D05E	53342					CHRCOUNT			
D05F	53343					SPRXSMSBS			
D060	53344					SCRNPTRLSB			
D061	53345					SCRNPTRMSB			
D062	53346					SCRNPTRBNK			
D063	53347	EXGLYPH	-		CHRCOUNT				SCRNPTRMB
D064	53348					COLPTRLSB			
D065	53349					COLPTRMSB			
D068	53352					CHARPTRLSB			
D069	53353					CHARPTRMSB			
D06A	53354					CHARPTRBNK			
D06B	53355					SPR16EN			
D06C	53356					SPRPTRADRLSB			
D06D	53357					SPRPTRADRMsb			
D06E	53358	SPRPTR16				SPRPTRBNK			
D06F	53359	PALNTSC	VGAHDTV			RASLINEO			
D070	53360	MAPEDPAL		BTPALSEL		SPRPALSEL		ABTPALSEL	
D071	53361				BP16ENS				
D072	53362				SPRYADJ				
D073	53363		RASTERHEIGHT						ALPHADELAY
D074	53364					SPRENALPHA			
D075	53365					SPRALPHAVAL			
D076	53366					SPRENV400			
D077	53367					SPRYMSBS			
D078	53368					SPRYSMSBS			
D079	53369					RASCMP			
D07A	53370	FNRST-CMP	EXTIROS		RESV	S PTR-CONT			RASCMPMSB
D07B	53371					DISPROWS			
D07C	53372	DEBUGC		VSYNCP	HSYNCP	RESV			BITPBANK

- **ABTPALSEL** VIC-IV bitmap/text palette bank (alternate palette)

- **ALPHADELAY** Alpha delay for compositor
- **ALPHEN** Alpha compositor enable
- **BBDRPOS** bottom border position
- **BITPBANK** Set which 128KB bank bitplanes
- **BORDERCOL** display border colour (256 colour)
- **BP16ENS** VIC-IV 16-colour bitplane enable flags
- **BTPALSEL** bitmap/text palette bank
- **CHARPTRBNK** Character set precise base address (bits 23 - 16)
- **CHARPTRLSB** Character set precise base address (bits 0 - 7)
- **CHARPTRMSB** Character set precise base address (bits 15 - 8)
- **CHR16** enable 16-bit character numbers (two screen bytes per character)
- **CHRCOUNT** Number of characters to display per row (LSB)
- **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)
- **COLPTRLSB** colour RAM base address (bits 0 - 7)
- **COLPTRMSB** colour RAM base address (bits 15 - 8)
- **DBLRR** When set, the Raster Rewrite Buffer is only updated every 2nd raster line, limiting resolution to V200, but allowing more cycles for Raster-Rewrite actions.
- **DEBUGC** VIC-IV debug pixel select red(01), green(10) or blue(11) channel visible in \$D07D
- **DISPROWS** Number of text rows to display
- **EXGLYPH** source full-colour character data from expansion RAM
- **EXTIRQS** Enable additional IRQ sources, e.g., raster X position.
- **FCLRHI** enable full-colour mode for character numbers >\$FF
- **FCLRLO** enable full-colour mode for character numbers <=\$FF
- **FNRASTERLSB** Read physical raster position
- **FNRASTERMSB** Read physical raster position
- **FNRST** Read raster compare source (0=VIC-IV fine raster, 1=VIC-II raster), provides same value as set in FNRSTCMP

- **FNRSTCMP** Raster compare is in physical rasters if clear, or VIC-II rasters if set
- **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
- **LINESTEPLSB** number of bytes to advance between each text row (LSB)
- **LINESTEPMSB** number of bytes to advance between each text row (MSB)
- **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)
- **NORRDEL** When clear, raster rewrite double buffering is used
- **PALEMU** Enable PAL CRT-like scan-line emulation
- **PALNTSC** NTSC emulation mode (max raster = 262)
- **RASCMP** Physical raster compare value to be used if FNRSTCMP is clear
- **RASCMPMSB** Raster compare value MSB
- **RASLINEO** first VIC-II raster line
- **RASTERHEIGHT** physical rasters per VIC-II raster (1 to 16)
- **RESV** Reserved.
- **RSTDELEN** Enable raster delay (delays raster counter and interrupts by one line to match output pipeline latency)
- **SCREENCOL** screen colour (256 colour)
- **SCRNPTRBNK** screen RAM precise base address (bits 23 - 16)
- **SCRNPTRLSB** screen RAM precise base address (bits 0 - 7)
- **SCRNPTRMB** screen RAM precise base address (bits 31 - 24)
- **SCRNPTRMSB** screen RAM precise base address (bits 15 - 8)
- **SDBDRWDLSB** Width of single side border (LSB)
- **SDBDRWDMSB** side border width (MSB)

- **SHDEMU** Enable simulated shadow-mask (PALEMU must also be enabled)
- **SMTH** video output horizontal smoothing enable
- **SPR16EN** sprite 16-colour mode enables
- **SPRALPHAVAL** Sprite alpha-blend value
- **SPRBPMEN** Sprite bitplane-modify-mode enables
- **SPRENALPHA** Sprite alpha-blend enable
- **SPRENV400** Sprite V400 enables
- **SPRH640** Sprite H640 enable
- **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)
- **SPRMC1** 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)
- **SPRPTRADRLSB** sprite pointer address (bits 7 - 0)
- **SPRPTRADRMSB** sprite pointer address (bits 15 - 8)
- **SPRPTRBNK** sprite pointer address (bits 23 - 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
- **SPRYADJ** Sprite Y position adjustment
- **SPRYMSBS** Sprite V400 Y position MSBs
- **SPRYSMSBS** Sprite V400 Y position super MSBs
- **SPTRCONT** Continuously monitor sprite pointer, to allow changing sprite data source while a sprite is being drawn
- **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 HDMI/HDTV VIC-II cycle-exact frame timing. May help to produce a functional display on older VGA monitors.
- **VSYNCP** vsync polarity
- **XPOSLSB** Read horizontal raster scan position LSB
- **XPOSMSB** Read horizontal raster scan position MSB

N

APPENDIX

Sound Interface Device (SID)

- SID Registers

SID REGISTERS

The MEGA65 has 4 SIDs build in, which can be access through the register ranges starting at \$D400, \$D420, \$D440, and \$D460. The registers in each of these ranges are exactly the same, so the following table only lists the first SID. Add 32 the get to the next SID respectively.

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D400	54272					VOICE1FRQLO			
D401	54273					VOICE1FRQHI			
D402	54274					VOICE1PWLO			
D403	54275			VOICE1UNSD				VOICE1PWHI	
D404	54276	VOICE1- CTRLRNW	VOICE1- CTRLPUL	VOICE1- CTRLSAW	VOICE1- CTRLTRI	VOICE1- CTRLTST	VOICE1- CTRLRMO	VOICE1- CTRLRMF	VOICE1- CTRLGATE
D405	54277		ENV1ATTDUR				ENV1DECDUR		
D406	54278		ENV1SUSDUR				ENV1RELDUR		
D407	54279				VOICE2FRQLO				
D408	54280					VOICE2FRQHI			
D409	54281					VOICE2PWLO			
D40A	54282		VOICE2UNSD					VOICE2PWHI	
D40B	54283	VOICE2- CTRLRNW	VOICE2- CTRLPUL	VOICE2- CTRLSAW	VOICE2- CTRLTRI	VOICE2- CTRLTST	VOICE2- CTRLRMO	VOICE2- CTRLRMF	VOICE2- CTRLGATE
D40C	54284		ENV2ATTDUR				ENV2DECDUR		
D40D	54285		ENV2SUSDUR				ENV2RELDUR		
D40E	54286				VOICE3FRQLO				
D40F	54287					VOICE3FRQHI			
D410	54288					VOICE3PWLO			
D411	54289		VOICE3UNSD					VOICE3PWHI	
D412	54290	VOICE3- CTRLRNW	VOICE3- CTRLPUL	VOICE3- CTRLSAW	VOICE3- CTRLTRI	VOICE3- CTRLTST	VOICE3- CTRLRMO	VOICE3- CTRLRMF	VOICE3- CTRLGATE
D413	54291		ENV3ATTDUR				ENV3DECDUR		
D414	54292		ENV3SUSDUR				ENV3RELDUR		
D415	54293				FLTRCUTFRQLO				
D416	54294				FLTRCUTFRQHI				
D417	54295		FLTRRESON			FLTR- EXTINP	FLTR- V1OUT	FLTR- V2OUT	FLTR- V3OUT
D418	54296	FLTR- CUTV3	FLTR- HIPASS	FLTR- BDPASS	FLTR- LOPASS			FLTRVOL	
D419	54297				PADDLE1				
D41A	54298				PADDLE2				
D41B	54299				OSC3RNG				

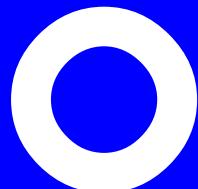
continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D41C	54300						ENV3OUT		
D63C	54844		-				SIDMODE		

- **ENV3OUT** Envelope Generator 3 Output
- **ENVXATTDUR** Envelope Generator X Attack Cycle Duration
- **ENVXDECDDUR** Envelope Generator X Decay Cycle Duration
- **ENVXRELDUR** Envelope Generator X Release Cycle Duration
- **ENVXSUSDUR** Envelope Generator X Sustain Cycle Duration
- **FLTRBDPASS** Filter Band-Pass Mode
- **FLTRCUTFRQHI** Filter Cutoff Frequency High
- **FLTRCUTFRQLO** Filter Cutoff Frequency Low
- **FLTRCUTV3** Filter Cut-Off Voice 3 Output (1 = off)
- **FLTREXTINP** Filter External Input
- **FLTRHIPASS** Filter High-Pass Mode
- **FLTRLOPASS** Filter Low-Pass Mode
- **FLTRRESON** Filter Resonance
- **FLTRVOL** Filter Output Volume
- **FLTRVXOUT** Filter Voice X Output
- **OSC3RNG** Oscillator 3 Random Number Generator
- **PADDLE1** Analog/Digital Converter: Game Paddle 1 (0-255)
- **PADDLE2** Analog/Digital Converter Game Paddle 2 (0-255)
- **SIDMODE** Select SID mode: 0=6581, 1=8580
- **VOICE1CTRLRMF** Voice 1 Synchronize Osc. 1 with Osc. 3 Frequency
- **VOICE1CTRLRMO** Voice 1 Ring Modulate Osc. 1 with Osc. 3 Output
- **VOICE2CTRLRMF** Voice 2 Synchronize Osc. 2 with Osc. 1 Frequency
- **VOICE2CTRLRMO** Voice 2 Ring Modulate Osc. 2 with Osc. 1 Output
- **VOICE3CTRLRMF** Voice 3 Synchronize Osc. 3 with Osc. 2 Frequency
- **VOICE3CTRLRMO** Voice 3 Ring Modulate Osc. 3 with Osc. 2 Output

- **VOICEXCTRLGATE** Voice X Gate Bit (1 = Start, 0 = Release)
- **VOICEXCTRLPUL** Voice X Pulse Waveform
- **VOICEXCTRLRNW** Voice X Control Random Noise Waveform
- **VOICEXCTRLSAW** Voice X Sawtooth Waveform
- **VOICEXCTRLTRI** Voice X Triangle Waveform
- **VOICEXCTRLTST** Voice X Test Bit - Disable Oscillator
- **VOICEXFRQHI** Voice X Frequency High
- **VOICEXFRQLO** Voice X Frequency Low
- **VOICEXPWHI** Voice X Pulse Waveform Width High
- **VOICEXPWLO** Voice X Pulse Waveform Width Low
- **VOICEXUNSD** Unused



APPENDIX

6526 Complex Interface Adaptor (CIA) Registers

- CIA 6526 Registers
- CIA 6526 Hypervisor Registers

CIA 6526 REGISTERS

CIA 1 Registers

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0					
DC00	56320	PORTA												
DC01	56321	PORTB												
DC02	56322	DDRA												
DC03	56323	DDRB												
DC04	56324	TIMERA												
DC05	56325	TIMERA												
DC06	56326	TIMERB												
DC07	56327	TIMERB												
DC08	56328	-			TODJIF									
DC09	56329	-		TODSEC										
DC0A	56330	-		TODMIN										
DC0B	56331	TOD-AMPM	-	TODHOUR										
DC0C	56332	SDR												
DC0D	56333	IR	ISRCLR		FLG	SP	ALRM	TB	TA					
DC0E	56334	TOD50	SPMOD	IMODA	-	RMODA	OMODA	PBONA	STRTA					
DC0F	56335	TODEDIT	IMODB		LOAD	RMODB	OMODB	PBONB	STRTB					

- **ALRM** TOD alarm
- **DDRA** Port A DDR
- **DDRB** Port B DDR
- **FLG** FLAG edge detected
- **IMODA** Timer A tick source
- **IMODB** Timer B tick source
- **IR** Interrupt flag
- **ISRCLR** Placeholder - Reading clears events
- **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
- **TODMIN** TOD minutes
- **TODSEC** TOD seconds

CIA2 Registers

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
DD00	56576								PORTA
DD01	56577								PORTB
DD02	56578								DDRA

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
DD03	56579					DDRB				
DD04	56580					TIMERA				
DD05	56581					TIMERA				
DD06	56582					TIMERB				
DD07	56583					TIMERB				
DD08	56584		-					TODJIF		
DD09	56585	-				TODSEC				
DD0B	56587	TOD-AMPM	-					TODHOUR		
DD0C	56588				SDR					
DD0D	56589	IR		ISRCLR		FLG	SP	ALRM	TB	TA
DD0E	56590	TOD50	SPMOD	IMODA	-	RMODA	OMODA	PBONA	STRTA	
DD0F	56591	TODEDIT		IMODB	LOAD	RMODB	OMODB	PBONB	STRTB	

- **ALRM** TOD alarm
- **DDR_A** Port A DDR
- **DDR_B** Port B DDR
- **FLG** FLAG edge detected
- **IMODA** Timer A tick source
- **IMODB** Timer B tick source
- **IR** Interrupt flag
- **ISRCLR** Placeholder - Reading clears events
- **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.

CIA 1 Hypervisor Registers

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
DC10	56336					TALATCH			
DC11	56337					TALATCH			
DC12	56338					TALATCH			

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
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	TOD-AMPM			TODHOUR				
DC1C	56348	DD00-DELAY			ALRMJIF				
DC1D	56349				ALRMSEC				
DC1E	56350				ALRMMIN				
DC1F	56351	ALRM-AMPM			ALRMHOUR				

- **ALRMAMPM** TOD Alarm AM/PM flag
- **ALRMHOUR** TOD Alarm hours value
- **ALRMJIF** TOD Alarm 10ths of seconds value (actually all 8 bits)
- **ALRMMIN** TOD Alarm minutes value
- **ALRMSEC** TOD Alarm seconds value
- **DD00DELAY** Enable delaying writes to \$DD00 by 3 cycles to match real 6502 timing
- **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

CIA2 Hypervisor Registers

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
DD10	56592								TALATCH
DD11	56593								TALATCH
DD12	56594								TALATCH
DD13	56595								TALATCH
DD14	56596								TALATCH
DD15	56597								TALATCH
DD16	56598								TALATCH
DD17	56599								TALATCH
DD18	56600	IMFLG	IMSP	IMALRM	IMTB				TODJIF
DD19	56601								TODSEC
DD1A	56602								TODMIN
DD1B	56603	TOD-AMPM							TODHOUR
DD1C	56604	DD00-DELAY							ALRMJIF
DD1D	56605								ALRMSEC
DD1E	56606								ALRMMIN
DD1F	56607	ALRM-AMPM							ALRMHOUR

- **ALRMAMPM** TOD Alarm AM/PM flag
- **ALRMHOUR** TOD Alarm hours value
- **ALRMJIF** TOD Alarm 10ths of seconds value (actually all 8 bits)
- **ALRMMIN** TOD Alarm minutes value
- **ALRMSEC** TOD Alarm seconds value
- **DD00DELAY** Enable delaying writes to \$DD00 by 3 cycles to match real 6502 timing
- **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

P APPENDIX

4551 UART, GPIO and Utility Controller

- C65 6551 UART Registers
- 4551 General Purpose I/O & Miscellaneous Interface Registers

C65 6551 UART REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0				
D600	54784	DATA											
D601	54785	-			FRMERR	PTYERR	RXOVR-RUN	RXRDY					
D602	54786	TXEN	RXEN	SYNCMOD		CHARSZ	PTYEN	PTYEVEN					
D603	54787	DIVISOR											
D604	54788	DIVISOR											
D605	54789	IMTXIRQ	IMRXIRQ	IMTXNMI	IMRXNMI	-							
D606	54790	IFTXIRQ	IFRXIRQ	IFTXNMI	IFRXNMI	-							

- **CHARSZ** UART character size: 00=8, 01=7, 10=6, 11=5 bits per byte
- **DATA** UART data register (read or write)
- **DIVISOR** UART baud rate divisor (16 bit). Baud rate = $7.09375\text{MHz} / \text{DIVISOR}$, unless MEGA65 fast UART mode is enabled, in which case baud rate = $80\text{MHz} / \text{DIVISOR}$
- **FRMERR** UART RX framing error flag (clear by reading \$D600)
- **IFRXIRQ** UART interrupt flag: IRQ on RX (not yet implemented on the MEGA65)
- **IFRXNMI** UART interrupt flag: NMI on RX (not yet implemented on the MEGA65)
- **IFTXIRQ** UART interrupt flag: IRQ on TX (not yet implemented on the MEGA65)
- **IFTXNMI** UART interrupt flag: NMI on TX (not yet implemented on the MEGA65)
- **IMRXIRQ** UART interrupt mask: IRQ on RX (not yet implemented on the MEGA65)
- **IMRXNMI** UART interrupt mask: NMI on RX (not yet implemented on the MEGA65)
- **IMTXIRQ** UART interrupt mask: IRQ on TX (not yet implemented on the MEGA65)
- **IMTXNMI** UART interrupt mask: NMI on TX (not yet implemented on the MEGA65)
- **PTYEN** UART Parity enable: 1=enabled
- **PTYERR** UART RX parity error flag (clear by reading \$D600)
- **PTYEVEN** UART Parity: 1=even, 0=odd
- **RXEN** UART enable receive
- **RXOVRRUN** UART RX overrun flag (clear by reading \$D600)
- **RXRDY** UART RX byte ready flag (clear by reading \$D600)

- **SYNCMOD** UART synchronisation mode flags (00=RX & TX both async, 01=RX sync, TX async, 1x=TX sync, RX async (unused on the MEGA65)
- **TXEN** UART enable transmit

4551 GENERAL PURPOSE I/O & 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	AC-CESSKEY	OSKDIM	REALHW		-		KEYUP	KEYLEFT
D610	54800					ASCIIEKEY			
D611	54801	MDISABLE	MCAPS	MSCRL	MALT	MMEGA	MCTRL	MLSHFT	MRSHTF
D612	54802	LJOYB	LJOYA	JOYSWAP	OSKDE-BUG				-
D615	54805	OSKEN				VIRTKEY1			
D616	54806	OSKALT				VIRTKEY2			
D617	54807	OSKTOP				VIRTKEY3			
D618	54808				KSCNRATE				
D619	54809				PETSCIIEKEY				
D61A	54810				SYSCTL				
D61D	54813	KEYLED-ENA				KEYLEDREG			
D61E	54814				KEYLEDVAL				
D620	54816				POTAX				
D621	54817				POTAY				
D622	54818				POTBX				
D623	54819				POTBY				
D625	54821				J21L				
D626	54822				J21H				
D627	54823				J21LDDR				
D628	54824				J21HDDR				

continued ...

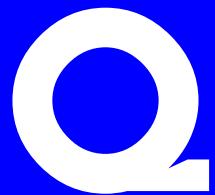
...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D629	54825								M65MODEL

- **ACCESSKEY** Enable accessible keyboard input via joystick port 2 fire button
- **ASCIIKEY** Last key press as ASCII (hardware accelerated keyboard scanner). Write to clear event ready for next.
- **BASHDDR** Data Direction Register (DDR) for \$D60D bit bashing port.
- **CONN41** Internal 1541 drive connect (1=connect internal 1541 drive to IEC bus)
- **HDSCL** HDMI I2C control interface SCL clock
- **HDSDA** HDMI I2C control interface SDA data line
- **J21H** J21 pins 11 – 14 input/output values
- **J21HDDR** J21 pins 11 – 14 data direction register
- **J21L** J21 pins 1 – 6, 9 – 10 input/output values
- **J21LDDR** J21 pins 1 – 6, 9 – 10 data direction register
- **JOYSWAP** Exchange joystick ports 1 & 2
- **KEYLEDENA** Keyboard LED control enable
- **KEYLEDREG** Keyboard LED register select (R,G,B channels x 4 = 0 to 11)
- **KEYLEDVAL** Keyboard LED register value (write only)
- **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)
- **M65MODEL** MEGA65 model ID. Can be used to determine the model of MEGA65 a programme is running on, e.g., to enable touch controls on MEGA-phone.
- **MALT** ALT key state (hardware accelerated keyboard nner – read only).
- **MCAPS** CAPS LOCK key state (hardware accelerated keyboard scanner – read only).

- **MCTRL** CTRL key state (hardware accelerated keyboard nner - read only).
- **MDISABLE** Disable modifiers (hardware accelerated keyboard scanner).
- **MLSHFT** Left shift key state (hardware accelerated keyboard nner - read only).
- **MMEGA** MEGA/C= key state (hardware accelerated keyboard nner - read only).
- **MRSHFT** Right shift key state (hardware accelerated keyboard nner - read only).
- **MSCRL** NOSCRL key state (hardware accelerated keyboard nner - read only).
- **OSKALT** Display alternate on-screen keyboard layout (typically dial pad for MEGA65 telephone)
- **OSKDEBUG** Debug OSK overlay (WRITE ONLY)
- **OSKDIM** Light or heavy dimming of background material behind on-screen keyboard
- **OSKEN** Enable display of on-screen keyboard composited overlay
- **OSKTOP** 1=Display on-screen keyboard at top, 0=Disply 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
- **PETSCIKEY** Last key press as PETSCII (hardware accelerated keyboard scanner). Write to clear event ready for next.
- **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.
- **REALHW** Set to 1 if the MEGA65 is running on real hardware, set to 0 if emulated (Xemu) or simulated (ghdl)
- **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
- **SYSCTL** System control flags (target specific)
- **UFAST** C65 UART BAUD clock source: 1 = 7.09375MHz, 0 = 80MHz (VIC-IV pixel clock)
- **VIRTKEY1** Set to \$7F for no key down, else specify virtual key press.
- **VIRTKEY2** Set to \$7F for no key down, else specify 2nd virtual key press.
- **VIRTKEY3** Set to \$7F for no key down, else specify 3nd virtual key press.



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 four receive buffers, allowing upto three frames to be received while another is being processed, or to allow less frequent processing of interrupts. These receive buffers can be memory mapped, and also directly accessed using the MEGA65's DMA controller. Together with automatic CRC32 checking on reception, and automatic CRC32 generation for transmit, these features considerably reduce the burden on the processor, and make it much simpler to write ethernet-enabled programs.

The 45E100 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. These disadvantages include the lack of working interrupts, as well as 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 megabytes 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 and TXRST bits (bit 0 of register \$D6E0) to ensure that the Ethernet controller is reset, and then set these bits to 1, to release the controller from the reset state. It will then auto-negotiate connection at the highest available speed, typically 100Mbit, full-duplex.

If you wish to simply poll for the arrival of ethernet frames, check the RXQ bit (bit 5 of \$D6E1). If it is set, then there is at least one frame that has been received. To access the next frame that has been received, write \$01 to \$D6E1, and then \$03 to \$D6E1. This will rotate the ring of receive buffers, to make the next received frame accessible by the processor. The receive buffer that was previously accessible by the processor is marked free, and the 45E100 will use it to receive another ethernet frame when required.

Because the 45E100 has four receive buffers, it is possible that to process multiple frames in succession by following this procedure. If all receive buffers contain received frames, and the processor has not accepted them, then the RXBLKD signal will be asserted, so that the processor knows that if any more frames are received, they will be lost. Programmers should take care to avoid this situation. As the 45E100 supports receive interrupts, this is generally easy to manage - but don't underestimate how often ethernet frames can arrive on a 100mbit Fast Ethernet connection: If a sender sends a continuous stream of minimum-length ethernet frames, they can arrive every 6 microseconds or so! While, it is unlikely that you will have to deal with such a high rate of packet reception, you should anticipate the need to process packets at least every milli-second. In particular, a once-per-frame CIA or raster IRQ may cause some packets to be lost, more than three arrive in a 16 - 20 ms video frame. The RXBLKD signal can be used to determine if this situation is likely to have occurred. But

note that it indicates only when all receive buffers are occupied, not if any further frames arrived while there were no free receive buffers.

The receive buffers are 2KB bytes each, and can each hold only one received ethernet frame at a time. This is different to some ethernet controllers that use their total receive buffer memory as a simple ring buffer. The reason for this is to keep the mechanism for programmers as simple as possible. By having the fixed buffers, it means that the controller can memory map the received ethernet frames in exactly the same location each time, making it possible to write much simpler receiver programs, because the location of the received ethernet frames can be assumed to be constant.

The structure of a receive buffer containing an ethernet frame is quite simple: The first two bytes indicate the length of the received frame. The frame then follows immediately. The effective Maximum Transport Unit (MTU) length is 2,042 bytes, as the last four bytes are occupied by the CRC32 checksum of the received ethernet frame. 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.
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 RXOPEN (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 5 microseconds. Fortunately, at the MEGA65's 40MHz full speed mode, and using the 20MB 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 I/O address space. Care must be taken as mapping the ethernet frame buffers into the \$D000 I/O address space causes all other I/O devices to unavailable during this time. Therefore CIA-based 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 Chapter/Appendix G on page G-11.

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 MEGA65's BASIC 65, 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 I/O 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 I/O 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 I/O 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 I/O 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("$")
```

The 45E100 Fast Ethernet controller has the following registers

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0							
D6E0	55008	TXIDLE	RXBLKD	-	RCENABLED	DRXDV	DRXD	TXRST	RST							
D6E1	55009	RXQEN	TXQEN	RXQ	TXQ	STRM	RXBF		-							
D6E2	55010	TXSZLSB														
D6E3	55011	TXSZMSB														
D6E4	55012	COMMAND														
D6E5	55013	RXPH		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)
- **MACADDRX** 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
- **RCENABLED** (read only) Ethernet remote control enable status
- **RST** Write 0 to hold ethernet controller under reset
- **RXBF** Number of free receive buffers
- **RXBLKD** Indicate if ethernet RX is blocked until RX buffers freed
- **RXPH** Ethernet RX clock phase adjust
- **RXQ** Ethernet RX IRQ status
- **RXQEN** Enable ethernet RX IRQ
- **STRM** Enable streaming of CPU instruction stream or VIC-IV display on ethernet
- **TXIDLE** Ethernet transmit side is idle, i.e., a packet can be sent.
- **TXPH** Ethernet TX clock phase adjust
- **TXQ** Ethernet TX IRQ status
- **TXQEN** Enable ethernet TX IRQ
- **TXRST** Write 0 to hold ethernet controller transmit sub-system under reset
- **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
```

HEX	DEC	Signal	Description
00	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.
01	1	STARTTX	Transmit packet
D0	208	RXNORMAL	Disable the effects of RXONLYONE
D4	212	DEBUGVIC	Select VIC-IV debug stream via ethernet when \$D6E1.3 is set
DC	220	DEBUGCPU	Select CPU debug stream via ethernet when \$D6E1.3 is set
DE	222	RXONLYONE	Receive exactly one ethernet frame only, and keep all signals states (for debugging ethernet sub-system)
F1	241	FRAME1K	Select 1KiB frames for video/cpu debug stream frames (for receivers that do not support MTUs of greater than 2KiB)
F2	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 states for in the MEGA65 Core repository on github in the src/tests and src/examples directories.

If you wish to use the ethernet controller for TCP/IP traffic, you may wish to examine the port of WeeIP to the MEGA65 at <https://github.com/mega65/mega65-weeip>. The code that controls the ethernet controller is located in eth.c.

R

APPENDIX

45IO27 Multi-Function I/O Controller

- Overview
- F011-compatible Floppy Controller
- SD card Controller and F011 Virtualisation Functions
- Touch Panel Interface
- Audio Support Functions
- Miscellaneous I/O Functions

OVERVIEW

The 45IO27 is a multi-purpose I/O controller that incorporates the functions of the C65's F011 floppy controller, together with the MEGA65's SD card controller interface, and a number of other miscellaneous I/O functions.

Each of these major functions is covered in a separate section of this chapter.

F011-COMPATIBLE FLOPPY CONTROLLER

The MEGA65 computer is one of the very few modern computers that still includes first-class support for magnetic floppy drives. It includes a floppy controller that is backwards compatible with the C65's F011D floppy drive controller.

However, unlike the F011D, the MEGA65's floppy disk controller supports HD and ED media, and similar to the 1541 floppy drive, it also supports variable data rates, so that a determined user could develop disk formats that store more data, include robust copy protection schemes, or both.

GCR encoding is not currently supported, but may be supported by a future revision of the controller. It may also be possible with some creativity and effort to use the debug register interface to read double-density GCR formatted media. This is because there are debug registers that can be queried to indicate the gap between each successive magnetic domain – which is sufficient to decode any disk format.

Multiple Drive Support

Like the C65's F011 floppy drive controller, the 45IO27 supports up to 8 drives. The first two of those drives, drive 0 and drive 1, are assumed to be connected to a standard 34-pin floppy cable, the same as used in standard PCs, i.e., with a twist in the cable to allow the use of two unjumpered drives.

As is described in later sections, it is possible to switch drive 0 and drive 1's position, without having to change cabling. Similarly, either or both of the first two drives may reference a real floppy drive, a D81 disk image stored on an attached SD card, or redirected to the floppy drive virtualisation service, so that the sector accesses can be handled by a connected computer, e.g., as part of a comfortable and efficient cross-development environment.

The remaining six drives are supported only in conjunction with a future C1565-compatible external drive port.

Buffered Sector Operations

The 45IO27 support two main modes of reading sectors from a disk: byte-by-byte, and via a memory-mapped sector buffer.

The byte-by-byte mechanism consists of having a loop wait for the DRO signal to be asserted, and then reading the byte of data from the DATA register (\$D087).

The memory-mapped sector buffer method consists of waiting for the BUSY flag to clear, indicating that the entire sector has been read, and then directly accessing the sector buffer located at \$FFD6C00 – \$FFD6dff. Care should be taken to ensure that the BUFSEL signal (bit 7 of \$D689) is cleared, so that the floppy sector buffer is visible, rather than the SD card sector buffer for programs other than the Hypervisor. This is because only the Hypervisor has access to the full 4KB SD controller buffer space: Normal programs see either the floppy sector buffer or the SD card sector buffer repeated 8 times between \$FFD6000 and \$FFD6FFF.

Alternatively, the sector buffer can be mapped at \$DE00 – \$DFFF, i.e., in the 4KB I/O area, by writing the \$81 to the SD command register at \$D680. This will hide any I/O peripherals that are otherwise using this area, e.g., from cartridges, or REU emulation. This function can be disabled again by writing \$82 to the SD command register. As with the normal sector buffer memory mapping at \$FFD6xxx, the BUFSEL signal (bit 7 of \$D689) affects whether the FDC or the SD card sector buffer is visible, for software not running in Hypervisor mode. Note that if you use the Matrix Mode / serial monitor interface to inspect the contents of the sector buffer, that this occurs in the Hypervisor context, and so the BUFSEL signal will be ignored, and the full 4KB buffer will be visible.

The memory-mapped sector buffer has the advantage that it can be accessed via DMA, allowing for very efficient copies. Also, it allows for loading a sector to occur in the background, while your program gets on with more interesting things in the meantime.

Reading Sectors from a Disk

There are several steps that you must follow in order to successfully read a sector from a disk. If you follow these instructions, your code will work with both physical disks, as well as D81 disk images that exist on the SD card:

- First, enable the motor and select the appropriate drive. The F011 supports upto 8 physical drives, although it is rare for more than two to be physically

connected. To enable the motor, write \$60 to \$D080. You should then write a SPINUP command (\$20) to \$D081, and wait for the BUSY flag (bit 7 of \$D082) to clear. The drive is now spinning at speed, and ready to service requests.

- Next, select the correct side of the disk by either setting or clearing the SIDE1 flag (bit 3 of \$D080). This takes effect immediately.
- Third, use the step-in and step-out commands (writing \$10 and \$18 to \$D081) as required to move the head to the correct track. Again, after each command, you should wait for the BUSY flag (bit 7 of \$D082) to clear, before issuing the next command.

Note that you can check if the head is at track 0 by checking the TRACK0 flag, but there is no fool-proof way to know if you are on any other specific track. You can use the registers at \$D6A3 - \$D6A5 to see the track, sector and side value from the last sector header which passed under the head to make an informed guess as to which track is currently selected. Note that this only works for real disks, as disk images do not spin under the read head. Also note that it is possible for tracks to contain sectors which purposely or accidentally have incorrect track numbers in the sector headers.

- Fourth, you need to load the desired track, sector and side number into the TRACK, SECTOR and SIDE registers (\$D084, \$D085 and \$D086, respectively). The FDC is now primed ready to read a sector.
- Fifth, you should write an appropriate read command value into \$D081. This will normally be \$40 (64). You then wait for the RDREQ signal (\$D083, bit 7) to go high, to indicate that the sector has been found. You then either wait for each occasion when DRQ goes high, and read byte-by-byte in such a loop, or wait for the BUSY flag to clear and the DRQ and EO flags to go high, which indicates that the complete sector has been read into the buffer.

Track Auto-Tune Function Deprecated

The 45IO27 also includes a track “auto-tune” function, which is enabled by clearing bit 7 of \$D696. That function reads the sector headers to determine which track the head is currently over, and steps the head in or out to try to get to the correct track. Auto-tune is enabled by default.

Sector Skew and Target Any Mode

It is also worth noting that the TARGANY signal can be asserted to tell the floppy controller to simply read the next sector that passes under the head. This applies only

when using real floppy disks, where it offers the considerable advantage of letting you read the sectors in the order in which they exist on the disk. This allows you to read a track at once, without having to wait for the index hole to pass by, or having to know which sector will next pass under the head.

For example, the C65 DOS formats disks using a skew factor of 7, while PCs may use a different skew-factor. If you don't know the skew factor of the disk, you may schedule the reading of the sectors on the track in a sub-optimal order. This can result in transfer rates as low as 5 sectors per second, compared with the optimal case of 50 sectors per second. Thus with either correct sector order, or using the target any mode, it is possible to read approximately two full tracks per second, i.e., two sides \times two tracks, or approximately 20KB/second on DD disks, or double that on HD disks, at around 40KB/second. This compares very favourably with the C65 DOS loading speed, which is typically nearer 1KB/sec in C64-mode.

Disk Layout and 1581 Logical Sectors

The 1581 disk format is unusual in that the physical sectors on the disk are a different size of the size of the data blocks that it presents to the user. Specifically, the disks use 512 byte sectors, while the 1581 (and C65) DOS present 256 byte data blocks. Two blocks are stored in each physical sector. Also, the physical track numbers are from 0 to 79, while the logical track numbers of the DOS are 1 to 80. Physical sectors are also numbered from 1 to 10, while logical block numbers begin are 0 to 39.

This means that if you want to find a 1581 logical sector, you need to know which physical sector it will be found in. To determine the physical sector that contains a block, you first subtract one from the track number, and then divide the sector number by two. Logical sectors 0 to 19 of each track are located in physical sectors 1 to 10 on the first side of the disk. Logical sectors 20 to 39 are located in physical sectors 1 to 10 on the reverse side of the disk.

Thus we can map a some logical track and sector t,s to the physical track, side and sector as follows:

$$track = t - 1$$

$$sector = (s/2) + 1, IFF s < 20, ELSE = ((s - 20)/2) + 1$$

$$side = 0 IFF sector < 20$$

It is also worth noting that the 45IO27 is capable of reading from tracks beyond track 80, provided that the disk drive is capable of this. Almost all 3.5 inch floppy drives are capable of reading at least one extra track, as historically manufacturers of floppy disks stored information about the disk on the 81st track. In our experience almost all drives will also be able to access an 82nd track.

FD2000 Disks

The CMD™ FD2000™ high-density 3.5" disk drives for Commodore™ computers use an unusual disk layout that is quite different from PCs: They use 10 sectors, the same as on 720KB double-density (DD) disks, but double the sector size from 512 bytes to 1,024 bytes. The 45IO27 does not currently support these larger sectors. At least read-only support is planned to be added via a core update in the future.

However, the 45IO27 *does* already support high-density disks and drives, with much higher capacities than the FD2000 was able to support.

High-Density and Variable-Density Disks

The 45IO27 supports variable data rates, allowing the use of HD drives and media, with a flexible approach to disk formats to support user experimentation, and the easy manipulation of high-capacity software distribution formats.

You are really only limited by your imagination, available time, and the limited number of people who are still interested in inserting a floppy disk into their computer!

The standard high-density (HD) disk format is "1.44MB", using 18 sectors per track over 80 tracks. This results in $80 \text{ tracks} \times 18 \text{ sectors} \times 2 \text{ sides} = 2,880 \text{ sectors}$. As each sector is 512 bytes, this corresponds to 1,440KB. This leads us into the interesting wonderland of "floppy disk marketing megabytes," a phenomena which long predates SD card and hard drive manufacturers using 1,000,000 byte megabytes.

Curiously for floppy disks, the 1,024,000 byte "megabyte" was used, i.e., "1MB" = $1\text{KB} \times 1\text{KB}$, that is a strange hybrid of binary and decimal conventions. Perhaps it was because the previous standard was 720KB, and they thought people would think it odd if double 720KB was 1.41MB, and complain about the missing kilo-bytes. We will continue to use the $1,024\text{KB} = 1,000\text{KB}$ floppy disk marketing mega-byte for consistency with this historical inconsistency.

However, HD floppy disks are fundamentally capable of holding much more than 1.44MB. For example, the FD2000 stored 1.6MB by using double-sized sectors to squeeze the equivalent of 20 sectors per track, and the Amiga went further by using track-at-once writing to fit 22 sectors per track. Both these formats used a constant data rate over all tracks, and thus a constant number of sectors per track.

However, the circumference of the tracks on a 3.5" floppy disk vary quite a lot: The inner track has a diameter of around 2.5cm, while the outside track is $1.6 \times$ longer. The 1.44MB disk format is designed so that the data is reliably stored on those shorter

inner tracks. This means that we should be able to fit 160% more data on the outermost track compared with the inner-most track, subject to a number of terms and conditions imposed by The Laws of Physics, the design of floppy drive electronics, the quality of media being used and various other annoying things. Because of this variability and uncertainty, the MEGA65's floppy controller supports fully variable data rate on a track-by-track basis.

Track Information Blocks

To support variable data rates, the 45GS27 supports the use of *Track Information Blocks* (TIBs) that contain information on the data rate and encoding used on the track. This allows users to experiment with various densities on various tracks, and yet have the disks function automatically for buffered sector operations.

The Track Information Block is automatically created when using the automatic track format function, but must be manually created if using unbuffered formatting. The TIB itself consists of the following data:

1. $3 \times \$A1$ Sync bytes (written with clock byte \$FB)
2. \$65 MEGA65 Track Information Block marker (written with clock byte \$FF, as are all following bytes in the block)
3. The track number
4. The data rate divisor, in the same format as \$D6A2, i.e., data rate = 40.5MHz / value.
5. Track encoding information: Bit 7 = Track-at-once flag, 1 = no inter-sector gaps (Amiga style), 0 = with inter-sector gaps (normal), Bit 6 = data encoding, 0 = MFM, 1=RLL2,7. Other bits are reserved, and should be 0 when written.
6. Sector count, i.e., number of sectors on the track.
7. CRC byte 1, using the normal floppy disk CRC algorithm.
8. CRC byte 2, using the normal floppy disk CRC algorithm.

The Track Information Block is always written at the data rate for a 720KB Double-Density disk, so that they can be present on any disk. Writing the Track Information Block and start-of-track gaps at the DD data rate also ensures that at very high data rates, the head still has sufficient time to switch to write mode, thus avoiding one of the many problems that arise when writing data at very high data rates.

If formatting disks unbuffered, it is the programmer's responsibility to switch the data rate after having written the Track Information Block, and several more bytes to allow

the floppy encoding pipeline to flush out the last byte of the Track Information Block. This is all automatically managed if using the automatic track formatting function.

The inclusion of the TIB allows users to play and explore the possibilities of different data rates on different drives and media, while still being automatically readable in all MEGA65s, because the TIB allows the controller to switch to the correct data rate and encoding. It is likely that over time somewhat standardised formats will develop, quite likely in the range of 2MB to 3.5MB – thus approaching the capacity of ED media in ED drives, without the need for those drives or media.

Formatting Disks

Formatting disks is now possible with the 45IO27, either unbuffered or fully-automatic. To format a track issue one of the following commands to \$D081:

- \$A0 – Automatic format, with inter-sector gaps, and write pre-compensation disabled.
- \$A1 – Manual format, write-precompensation disabled.
- \$A4 – Automatic format, with inter-sector gaps, and write pre-compensation enable.
- \$A5 – Manual format, write-precompensation enabled.
- \$A8 – Automatic format, Amiga-style track-at-once, and write pre-compensation disabled.
- \$AC – Automatic format, Amiga-style track-at-once, and write pre-compensation enable.

Manual formatting is not recommended, unless mastering track-at-once formatted disks for software distribution, because of the relative complexity of doing so. Also, at the higher data rates, bytes have to be delivered to the floppy controller as often as every 20 cycles, which requires considerable care when writing the format routine. For more information on manual formatting tracks, refer to the C64 Specifications Manual or the C65 ROM DOS source code, for examples of manual formatting.

The automatic modes, in contrast, format a track with a single command, and are thus much easier to use, and are recommended for general use. Write pre-compensation should normally be enabled, as it is required at higher data rates, and does not cause problems at lower data rates.

Write Pre-Compensation

Write pre-compensation is a family of algorithms used when writing high data-rate signals to floppy disks. It is used to anticipate and cancel out the predictable component of timing variation of magnetic recording. There are a variety of sources of this timing variation, which have been the subject of PhD theses, and a lot of proprietary research by hard drive manufacturers. What is important for us to understand is that adjacent pulses (really magnetic inversions) get pushed together, if they are surrounded by longer pulses, or tend to spread apart if surrounded by shorter pulses.

There are also other fascinatingly complex and difficult to predict factors, that cause things such as the “negative shift of mid-length pulses”, “inverse F-distribution of pulse arrival times” and goodness knows what else. But we shall leave those to the hard drive manufacturers. We limit ourselves to the data pattern induced effect described in the previous paragraph.

The 45GS27 supports two tunable coefficients for small and large corrections to this, which are used with an internal look-up table. However, this is all automatically handled if you enable write pre-compensation. This allows data rates that much more closely approach the expected limit of HD media, although due to the other horrors of magnetic media recording alluded to above, the actual limit is not reached.

Buffered Sector Writing

The 45IO27 can write to disk images that are located on the SD card, or when using virtualised disk access.

To write a sector, you follow a similar process to reading, except that you write \$84 to the command byte instead of \$40. The \$80 indicates a write, and the \$04 activates write-precompensation. This is important when writing to real floppy disks, especially HD and ED disks. Write-precompensation causes bits to be written slightly early or slightly late, using an algorithm that models how the magnetic domains on a disk tend to move after being written.

If you do not wish to use the sector buffer, but instead provide each byte one at a time during the write operation, you must add \$01 to the command code. However, this is not recommended on the MEGA65, because when writing to the SD card or using virtualised disk images the entire sector operation can happen instantaneously from the perspective of your program. This means that it is not possible to supply data reliably when in this mode. Thus apart from being less convenient, it is also less reliable.

Once a write operation has been triggered, the DRQ signal indicates when you should provide the next byte if performing a byte-by-byte write. Otherwise, it is assumed

that you will have pre-filled the sector buffer with the complete 512 bytes of data required.

To write to disks that contain Track Information Blocks, you should first wait for the TIB to be read when changing tracks. This is done by waiting for \$D6A9 (sectors per track from the TIB) to contain a non-zero value.

F011 Floppy Controller Registers

The following are the set of F011 compatibility registers of the 45IO47. Note that registers related to the use of SD card based storage are found in the corresponding section below.

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D080	53376	IRQ	LED	MOTOR	SWAP	SIDE			DS
D081	53377	WRCMD	RDCMD	FREE	STEP	DIR	ALGO	ALT	NOBUF
D082	53378	BUSY	DRQ	EQ	RNF	CRC	LOST	PROT	TK0
D083	53379	RDREQ	WTREQ	RUN	WGATE	DISKIN	INDEX	IRQ	DSKCHG
D084	53380				TRACK				
D085	53381				SECTOR				
D086	53382				SIDE				
D087	53383				DATA				
D088	53384				CLOCK				
D089	53385				STEP				
D08A	53386				PCODE				

- **ALGO** Selects reading and writing algorithm (currently ignored).
- **ALT** Selects alternate DPLL read recovery method (not implemented)
- **BUSY** F011 FDC busy flag (command is being executed) (read only)
- **CLOCK** Set or read the clock pattern to be used when writing address and data marks. Should normally be left \$FF
- **COMMAND** F011 FDC command register
- **CRC** F011 FDC CRC check failure flag (read only)
- **DATA** F011 FDC data register (read/write) for accessing the floppy controller's 512 byte sector buffer
- **DIR** Sets the stepping direction (inward vs
- **DISKIN** F011 Disk sense (read only)

- **DRQ** F011 FDC DRQ flag (one or more bytes of data are ready) (read only)
- **DS** Drive select (0 to 7). Internal drive is 0. Second floppy drive on internal cable is 1. Other values reserved for C1565 external drive interface.
- **DSKCHG** F011 disk change sense (read only)
- **EQ** F011 FDC CPU and disk pointers to sector buffer are equal, indicating that the sector buffer is either full or empty. (read only)
- **FREE** Command is a free-format (low level) operation
- **INDEX** F011 Index hole sense (read only)
- **IRQ** The floppy controller has generated an interrupt (read only). Note that interrupts are not currently implemented on the 45GS27.
- **LED** Drive LED blinks when set
- **LOST** F011 LOST flag (data was lost during transfer, i.e., CPU did not read data fast enough) (read only)
- **MOTOR** Activates drive motor and LED (unless LED signal is also set, causing the drive LED to blink)
- **NOBUF** Reset the sector buffer read/write pointers
- **PCODE** (Read only) returns the protection code of the most recently read sector. Was intended for rudimentary copy protection. Not implemented.
- **PROT** F011 Disk write protect flag (read only)
- **RDCMD** Command is a read operation if set
- **RDREQ** F011 Read Request flag, i.e., the requested sector was found during a read operation (read only)
- **RNF** F011 FDC Request Not Found (RNF), i.e., a sector read or write operation did not find the requested sector (read only)
- **RUN** F011 Successive match. A synonym of RDREQ on the 45IO47 (read only)
- **SECTOR** F011 FDC sector selection register
- **SIDE** Directly controls the SIDE signal to the floppy drive, i.e., selecting which side of the media is active.
- **STEP** Writing 1 causes the head to step in the indicated direction
- **SWAP** Swap upper and lower halves of data buffer (i.e. invert bit 8 of the sector buffer)
- **TKO** F011 Head is over track 0 flag (read only)

- **TRACK** F011 FDC track selection register
- **WGATE** F011 write gate flag. Indicates that the drive is currently writing to media. Bad things may happen if a write transaction is aborted (read only)
- **WRCMD** Command is a write operation if set
- **WTREQ** F011 Write Request flag, i.e., the requested sector was found during a write operation (read only)

The following registers apply to the 45IO27 only, i.e., are MEGA65 specific:

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D6A0	54944	DENSITY	DBGMO-TORA	DBGMO-TORA	DBGDIR	DBGDIR	DBGW-DATA	DBGW-GATE	DBGW-GATE
D6A2	54946					DATARATE			

- **DATARATE** Set number of bus cycles per floppy magnetic interval (decrease to increase data rate)
- **DBGDIR** Control floppy drive STEPDIR line
- **DBGMOTORA** Control floppy drive MOTOR line
- **DBGWDATA** Control floppy drive WDATA line
- **DBGWGATE** Control floppy drive WGATE line
- **DENSITY** Control floppy drive density select line

SD CARD CONTROLLER AND F011 VIRTUALISATION FUNCTIONS

For those situations where you do not wish to use real floppy disks, the 45IO27 supports two complementary alternative modes:

- SD card Based Disk Image Access.
- Virtualised Disk Image Access.

This is in addition to providing direct access to a dual-bus SD card interface.

SD card Based Disk Image Access

The 45IO27 is both a floppy drive and SD card controller. This enables it to transparently allow access to D81 disk images stored on the SD card. Further, because the controller is combined, it is possible to still have the floppy drive step and spin as though it were being used, providing considerable atmosphere and sense of realism, even when using disk images.

The 45IO27 supports both 800KB standard D81 disk images, as well as 64MB "MEGA Images". While an operating system may impose restrictions based on the name of a file, the 45IO27 is blind to these requirements. Instead, it requires only that a contiguous 800KB or 64MB of the SD card is used to contain a disk image.

When a disk image is enabled, the corresponding set of sectors on the SD card are effectively placed under user control, and the operating system is no longer able to prevent the reading or writing of any of those sectors. Thus you should never enable access to an image that is shorter than the required size, as it will otherwise allow the user to unwittingly or maliciously access and/or modify data that is not part of the image file.

For the same reason, only the hypervisor can change the sector number where a disk image starts (the D?STARTSEC? signals), or allow the use of disk images instead of the real floppy drive (USEREAL0 and USEREAL1 signals). Once the Hypervisor has set the start sector of a disk image, and cleared the USEREAL0 or USEREAL1 signal, the user can still control whether an access will go to the real floppy drive or to the disk image by respectively clearing or setting the appropriate signal. For drive 0, this is D0IMG, and for drive 1, it is D1IMG.

There are also signals to control whether a disk image is an 800KB D81 image or a 64MB MEGA Disk image, and whether a disk image is present, and whether it is write protected. These are all located in the \$D68B register. Because of the ability of manipulation of these registers to corrupt or improperly access data, these signals are all read-only, except from within the hypervisor.

The following table lists the registers that are used to control access to disk images resident on the SD card:

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D68A	54922	D1D64	D0D64						-
D68B	54923	D1MD	D0MD	D1WP	D1P	D1IMG	D0WP	D0P	D0IMG
D68C	54924					D0STARTSEC0			
D68D	54925					D0STARTSEC1			
D68E	54926					D0STARTSEC2			
D68F	54927					D0STARTSEC3			

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D690	54928					D1STARTSEC0			
D691	54929					D1STARTSEC1			
D692	54930					D1STARTSEC2			
D693	54931					D1STARTSEC3			
D6A1	54945		-			SILENT	USE- REAL1	TARGANY	USE- REAL0

- **DOD64** F011 drive 0 disk image is D64 mega image if set (otherwise 800KiB 1581 or D65 image)
- **DOIMG** F011 drive 0 use disk image if set, otherwise use real floppy drive.
- **DOMD** F011 drive 0 disk image is D65 image if set (otherwise 800KiB 1581 image)
- **DOP** F011 drive 0 media present
- **DOSTARTSEC0** F011 drive 0 disk image address on SD card (LSB)
- **DOSTARTSEC1** F011 drive 0 disk image address on SD card (2nd byte)
- **DOSTARTSEC2** F011 drive 0 disk image address on SD card (3rd byte)
- **DOSTARTSEC3** F011 drive 0 disk image address on SD card (MSB)
- **DOWP** Write enable F011 drive 0
- **D1D64** F011 drive 1 disk image is D64 image if set (otherwise 800KiB 1581 or D65 image)
- **D1IMG** F011 drive 1 use disk image if set, otherwise use real floppy drive.
- **D1MD** F011 drive 1 disk image is D65 image if set (otherwise 800KiB 1581 image)
- **D1P** F011 drive 1 media present
- **D1STARTSEC0** F011 drive 1 disk image address on SD card (LSB)
- **D1STARTSEC1** F011 drive 1 disk image address on SD card (2nd byte)
- **D1STARTSEC2** F011 drive 1 disk image address on SD card (3rd byte)
- **D1STARTSEC3** F011 drive 1 disk image address on SD card (MSB)
- **D1WP** Write enable F011 drive 1
- **SILENT** Disable floppy spinning and tracking for SD card operations.

- **TARGANY** Read next sector under head if set, ignoring the requested side, track and sector number.
- **USEREAL0** Use real floppy drive for drive 0 if set (read-only, except for from hypervisor)
- **USEREAL1** Use real floppy drive for drive 1 if set (read-only, except for from hypervisor)

F011 Virtualisation

In addition to allowing automatic read and write access to SD card based D81 images, it is possible to connect a program to the serial monitor interface that provides and accepts data as though it were the floppy disk.

This is commonly used in a cross-development environment, where you wish to frequently modify a disk image that is used by a program you are developing – without the need to continually push new versions of the disk image on the MEGA65’s SD card first. It also has the added benefit that it allows you to easily visualise which sectors are being read from and written to, which can help speed up development and debugging of your program.

This function operates together with the MEGA65’s Hypervisor by triggering hyperrupts (that is, interrupts that activate the Hypervisor). There is then special code in the Hypervisor that communicates with the `m65` program via the serial monitor interface.

If that all sounds rather complex, all you need to know is that to use this function, you run the `m65` utility with arguments like `-d image.d81`. This should automatically establish the link with the MEGA65. If the BASIC interpreter stops responding, press the reset button (not the power switch) on the left-hand side of your MEGA65, and it should return to the BASIC’s READY. prompt – and if your supplied disk image has a C65 auto-boot function, then it should automatically start booting.

This function works very well if the host computer runs Linux, and will allow loading at a speed of around 60KB per second. However, it may be much slower on Windows or Apple OSX-based systems.

Of course to use this, you will also need an interface module and/or cable to connect your cross-development system to the MEGA65’s serial monitor interface. This is most easily done using a Trenz TE0790-03 JTAG adapter and mini-USB cable.

More information on using this interface and the `m65` tool can be found in Chapter/Appendix [14 on page 14-3](#).

Dual-Bus SD card Controller

The 45IO27 contains a high-speed dual-bus SD card controller. This controller operates in SPI x1 mode at a clock speed of 20MHz, providing a maximum throughput of approximately 2MB/sec. The quality of the SD card makes a significant difference to performance, with some cards routinely delivering 1.7MB/sec, while others 1MB/sec or less. Generally speaking, newer cards marketed as being suitable for video recording perform better. The controller supports SDHC cards, and has experimental support for SDXC cards. Legacy SD cards with a capacity of 2GB or less are not supported, as these use a different addressing mode.

The SD controller itself is very simple to drive: Supply the sector number in \$D861-\$D684, and then issue a read or write command to the command register (\$D680). The SD controller supports only sector-based buffered operations, using the sector buffer. In hypervisor mode, the sector buffer is located at \$FFD6E00 - \$FFD6FFF, while when the computer is in normal operating mode, the SD card and the floppy controller share a single address for both the floppy drive and SD card sector buffers. Which buffer is visible at that address is dictated by the BUFSEL signal. If it is 1, then the SD card buffer is visible, while if it is 0, then the floppy drive sector buffer is visible. See also Sub-section [R on page R-4](#) for further discussion on the precise behaviour of this buffer with regard to normal mode versus Hypervisor mode, and how it can also be mapped at \$DE00.

Write Gate

When writing a sector, you must, however, first open the “write gate”. This is a mechanism to prevent accidental corruption of data on the SD card, as it requires two different values to be written to the command register (\$D680) in quick succession: You have approximately 1 milli second after opening the write gate to command the write, before the write gate effectively closes again, write-protecting the SD card until the write gate is opened again. There are two different write gates: One for the master boot record (sector 0), and the other for all other sectors, both of which are listed in the command table below. This is designed to provide additional protection to the very important master boot record sector against programs accidentally calculating sector 0 as the target for an ordinary write.

Fill Mode

Where you wish to fill sectors with a constant value, the 45IO27 supports a mode for this, so that you do not need to overwrite the contents of the sector buffer. This is

activated by placing the desired fill value into the FILLVAL register (\$D686), and then issuing the enable fill mode command (\$83), performing the sector write operations, and then issuing the disable fill mode command (\$84).

Selecting Among Multiple SD cards

The controller supports two SD card interfaces, and it is possible to have a card in both at the same time. However, each card needs to be reset and commanded separately. Only one card can be commanded at a time. That said, it is possible to reset each card once, and then switch between the cards to perform individual operations.

To select the first SD card slot, write \$C0 to the SD Controller Command Register (\$D680). To select the second SD card slot, write \$C1 instead.

SD Controller Command Table

The SD controller supports the following commands that can be written to the command register at \$D680:

Command	Function
\$00 (0)	Place SD card under reset (deprecated. Use command \$10 instead)
\$01 (1)	Release SD card from reset
\$02 (2)	Read a sector from the SD card
\$03 (3)	Write a single sector to the SD card
\$04 (4)	Write the first sector of a multi-sector write to the SD card
\$05 (5)	Write a subsequent sector of a multi-sector write to the SD card
\$06 (6)	Write the final sector of a multi-sector write to the SD card
\$0C (12)	Request flush of SD card write buffers (experimental)
\$0E (14)	Pull SD handshake line low (debug only)
\$0F (15)	Pull SD handshake line high (debug only)
\$10 (16)	Place SD card under reset with flags set (preferred method)
\$11 (17)	Release SD card from reset (alternate method)
\$40 (64)	Clear the SDHC/SDXC flag, selecting legacy SD card mode (deprecated)

continued ...

...continued

Command	Function
\$41 (65)	Set the SDHC/SDXC mode flag
\$44 (68)	End force clearing of SD card state machine error flag
\$45 (69)	Begin force clearing of SD card state machine error flag
\$4D (77)	Open write-gate to sector 0 (master boot record) for approximately 1 milli-second
\$57 (87)	Open write-gate for all sectors > 0 for approximately 1 milli-second
\$81 (129)	Enable mapping of the SD/FDC sector buffer at \$DE00 - \$DFFF
\$82 (130)	Disable mapping of the SD/FDC sector buffer at \$DE00 - \$DFFF
\$83 (131)	Enable SD card Fill Mode
\$84 (132)	Disable SD card Fill Mode
\$C0 (192)	Select SD card Slot 0
\$C1 (193)	Select SD card Slot 1

Note that the hypervisor can enable or disable direct access to the SD controller. The hypervisor operating system may provide a mechanism for requesting permission to access the SD card controller, e.g., for disk management utilities.

The SD card controller registers are as follows:

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D680	54912	CMDANDSTAT							
D681	54913	SECTOR0							
D682	54914	SECTOR1							
D683	54915	SECTOR2							
D684	54916	SECTOR3							
D686	54918	FILLVAL							
D68A	54922	-				VFDC1	VFDC0	VICIII	CDC00
D6AE	54958	FDCTIBEN	FDC-2XSEL	FDC-VARSPD	AUTO-2XSEL	FDCENC			
D6AF	54959	-	VLOST	VDRQ	VRNF	VEQINH	VW-FOUND	VRFOUND	

- **AUTO2XSEL** Automatically select DD or HD decoder for last sector display
- **CDC00** (read only) Set if colour RAM at \$DC00

- **CMDANDSTAT** SD controller status/command
- **FDC2XSEL** Select HD decoder for last sector display
- **FDCENC** Select floppy encoding (0=MFM, 1=RLL2,7, F=Raw encoding)
- **FDCTIBEN** Enable use of Track Info Block settings
- **FDCVARSPD** Enable automatic variable speed selection for floppy controller using Track Information Blocks on MEGA65 HD floppies
- **FILLVAL** WRITE ONLY set fill byte for use in fill mode, instead of SD buffer data
- **SECTOR0** SD controller SD sector address (LSB)
- **SECTOR1** SD controller SD sector address (2nd byte)
- **SECTOR2** SD controller SD sector address (3rd byte)
- **SECTOR3** SD controller SD sector address (MSB)
- **VDRQ** Manually set f011_drq signal (indented for virtual F011 mode only)
- **VEQINH** Manually set f011_eq_inhibit signal (indented for virtual F011 mode only)
- **VFDC0** (read only) Set if drive 0 is virtualised (sectors delivered via serial monitor interface)
- **VFDC1** (read only) Set if drive 1 is virtualised (sectors delivered via serial monitor interface)
- **VICIII** (read only) Set if VIC-IV or ethernet IO bank visible
- **VLOST** Manually set f011_lost signal (indented for virtual F011 mode only)
- **VRFOUND** Manually set f011_rsector_found signal (indented for virtual F011 mode only)
- **VRNF** Manually set f011_rnf signal (indented for virtual F011 mode only)
- **VWFOUND** Manually set f011_wsector_found signal (indented for virtual F011 mode only)

TOUCH PANEL INTERFACE

Some MEGA65 variants include an LCD touch panel, primarily the MEGAphone hand-held version of the MEGA65. The touch interface supports the detection of two simultaneous touch events. Some variants may also support gesture detection, however, this is still very experimental.

The touch detection interface that is contained in the 45IO27 is complemented by the on-screen-keyboard interface of the 4551 UART and GPIO controller. Refer to section P for further information. Of particular relevance are bit 7 of the registers \$D615 - \$D617 which allow activating the on-screen keyboard interface, selecting whether the on-screen keyboard is placed in the upper or lower portion of the screen, and whether the primary or secondary on-screen keyboard is displayed.

Direct connections between the 4551 and the 45IO27 combine information about any currently displayed on-screen keyboard and the touch interface controller, allowing synthetic keyboard events to be automatically triggered when the on-screen keyboard portion of the touch interface is pressed. This allows the touch interface to be used to drive the on-screen keyboard without requiring any support from user programs. This works even when the on-screen keyboard is moving during activation or transitioning between the top and bottom of the screen.

As touch interfaces can require calibration, the 45IO27 allows for a linear transformation of both the X and Y coordinates of a touch event. Specifically, there are scale (TCHXSCALE and TCHYSCALE) and offset registers (TCHXDELTA and TCHYDELTA) that provide for this transformation. It is also possible to flip the touch screen coordinates in either or both the X and Y axes. These calibration registers also affect the operation of the on-screen keyboard.

It should also be noted that some touch interfaces do not have constant horizontal or vertical resolution. For example, some panels have a low horizontal resolution region in the middle of the panel, which can require some care to accommodate.

To detect the primary touch event, the TOUCH1XLSB, TOUCH1XMSB, TOUCH1YLSB, TOUCH1YMSB registers can be read. Similar registers exist for the 2nd touch event: TOUCH2XLSB, TOUCH2XMSB, TOUCH2YLSB, TOUCH2YMSB. Each touch event has a single bit flag that indicates whether the touch event is currently valid: the EV1 and EV2 bits of the register \$D6B0. There are also corresponding bit-fields that indicate whether a given touch event has been made or released, allowing the detection of when a finger both makes and breaks contact with the screen. The UPDN1 and UPDN2 signals provide this information. Binary values of 01 and 10, respectively indicate if the finger has been removed or pressed against the touch panel. Values of 00 and 11 mean that a finger is either being held or not being held against the touch panel.

The primary touch event is also fed into the lightpen input of the VIC-IV, and can be detected using the normal light pen registers of the VIC-IV.

The registers for the touch panel interface are as follows:

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D6B0	54960	YINV	XINV		UPDN2		UPDN1	EV2	EV1

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D6B1	54961				CALXSCALELSB				
D6B2	54962				CALXSCALEMSB				
D6B3	54963				CALYSCALELSB				
D6B4	54964				CALYSCALEMSB				
D6B5	54965				CALXDELTALSB				
D6B7	54967				CALYDELTALSB				
D6B8	54968				CALYDELTAMSB				
D6B9	54969				TOUCH1XLSB				
D6BA	54970				TOUCH1YLSB				
D6BB	54971	-		TOUCH1YMSB		-		TOUCH1XMSB	
D6BC	54972				TOUCH2XLSB				
D6BD	54973				TOUCH2YLSB				
D6BE	54974	-		TOUCH2YMSB		-		TOUCH2XMSB	
D6C0	54976		GESTUREID				GESTUREDIR		

- **CALXDELTALSB** Touch pad X delta LSB
- **CALXSCALELSB** Touch pad X scaling LSB
- **CALXSCALEMSB** Touch pad X scaling MSB
- **CALYDELTALSB** Touch pad Y delta LSB
- **CALYDELTAMSB** Touch pad Y delta MSB
- **CALYSCALELSB** Touch pad Y scaling LSB
- **CALYSCALEMSB** Touch pad Y scaling MSB
- **EV1** Touch event 1 is valid
- **EV2** Touch event 2 is valid
- **GESTUREDIR** Touch pad gesture directions (left,right,up,down)
- **GESTUREID** Touch pad gesture ID
- **TOUCH1XLSB** Touch pad touch #1 X LSB
- **TOUCH1XMSB** Touch pad touch #1 X MSBs
- **TOUCH1YLSB** Touch pad touch #1 Y LSB
- **TOUCH1YMSB** Touch pad touch #1 Y MSBs
- **TOUCH2XLSB** Touch pad touch #2 X LSB
- **TOUCH2XMSB** Touch pad touch #2 X MSBs

- **TOUCH2YLSB** Touch pad touch #2 Y LSB
- **TOUCH2YMSB** Touch pad touch #2 Y MSBs
- **UPDN1** Touch event 1 up/down state
- **UPDN2** Touch event 2 up/down state
- **XINV** Invert horizontal axis
- **YINV** Invert vertical axis

AUDIO SUPPORT FUNCTIONS

The 45IO27 provides the primary interface into the MEGA65's full cross-bar audio mixer. This includes the interface for reading or modifying the mixer co-efficients, as well as accessing the mixer feedback registers, and setting the 16-bit digital sample values that are two of the input channels into the audio mixer.

The audio mixer consists of 128 coefficients, each of which is 16 bits. Each audio output channel, e.g., left speaker, right speaker, left headphone, right headphone, cellular modem 1 (MEGAphone models only) and so on, are generated by taking each of the audio input channels, multiplying them by the appropriate coefficient, and adding it to the total output of the audio output channel.

Because each audio output channel has its own set of coefficients that are applied to all of the audio input channels, this means that it is possible to produce totally different audio out each audio channel: For example, it is possible to play your favourite quadraphonic SID music out of the headphones while rick-rolling passers by with Amiga-style MOD audio. This is why the audio mixer is referred to as a **full cross-bar** mixer, because there are no restrictions on how you mix each audio output channel. In this regard, it is very similar to a full-function audio desk, allowing different mixing levels for different speakers.

Because the audio coefficients are 16 bits each, each one is formed using two successive bytes of the audio co-efficient space. Changes to the audio coefficients take effect immediately, so care should be taken when changing coefficients to avoid audible clicks and pops. Also, you must allow 32 cycles to elapse before changing the selected audio coefficient, as otherwise the change may be discarded if the audio mixer accumulator has not had time to re-visit that coefficient.

The audio sources on the MEGA65 and MEGAphone devices are as follows:

Input Channel ID	Connection
\$0 (0)	Left SIDs
\$1 (1)	Right SIDs
\$2 (2)	Modem Bay 1 (MEGApone only)
\$3 (3)	Modem Bay 2 (MEGApone only)
\$4 (4)	Bluetooth™ Left
\$5 (5)	Bluetooth™ Right
\$6 (6)	Headphone Interface 1
\$7 (7)	Headphone Interface 2
\$8 (8)	Digital audio Left
\$9 (9)	Digital audio Right
\$A (10)	MEMs Microphone 0 (Nexys4 and MEGApone only)
\$B (11)	MEMs Microphone 1 (MEGApone only)
\$C (12)	MEMs Microphone 2 (MEGApone only)
\$D (13)	MEMs Microphone 3 (MEGApone only)
\$E (14)	Headphone jack microphone (Nexys4 and MEGApone only)
\$F (15)	OPL-compatible FM audio (<i>shares co-efficient with input 14</i>)

The OPL-compatible FM audio which is on source 15 is controlled by the coefficient for source 14. This is because the coefficient for source 15 provides the master volume level for each output.

The audio cross-bar mixer supports the following eight output channels:

Output Channel ID	Connection
\$0 (0)	Left Primary Speaker (digital audio on MEGA65 R2/R3, physical speaker on MEGApone, headphone jack audio on Nexys4)
\$1 (1)	Right Primary Speaker (digital audio on MEGA65 R2/R3, physical speaker on MEGApone, headphone jack audio on Nexys4)
\$2 (2)	Modem Bay 1 audio output (MEGApone only)
\$3 (3)	Modem Bay 2 audio output (MEGApone only)
\$4 (4)	Bluetooth Left Audio (MEGApone only)
\$5 (5)	Bluetooth Right Audio (MEGApone only)

continued ...

...continued

Output Channel ID	Connection
\$6 (6)	Headphone Left output (MEGA65 R2/R3 and MEGAphone only. On Nexys4 boards the primary speaker drives the 3.5mm jack)
\$7 (7)	Headphone Right output (MEGA65 R2/R3 and MEGAphone only. On Nexys4 boards the primary speaker drives the 3.5mm jack)

To determine the coefficient register number for a given source and output, multiply the output number by 32 and multiply the source number by 2. This will be the register number for the LSB of the 16-bit coefficient. The MSB will be the next register. For example, to set the coefficient of the right SIDs to the 2nd modem bay audio output, the coefficient would be $32 \times 3 + 1 \times 2 = 96 + 2 = 98$.

XXX - mixer stuff XXX - mixer feedback registers XXX - Left and right digi XXX - CPU register for selecting PWM/PDM

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D6F4	55028					MIXREGSEL			
D6F5	55029					MIXREGDATA			
D6F8	55032					DIGILSB			
D6F9	55033					DIGILMSB			
D6FA	55034					DIGIRLSB			
D6FB	55035					DIGIRMSB			
D6FC	55036					READBACKLSB			
D6FD	55037					READBACKMSB			
D711	55057		-			PWMPDM		-	

- **DIGILEFTLSB** Digital audio, left channel, LSB
- **DIGILEFTMSB** Digital audio, left channel, MSB
- **DIGILSB** 16-bit digital audio out (left LSB)
- **DIGILMSB** 16-bit digital audio out (left MSB)
- **DIGIRIGHTLSB** Digital audio, left channel, LSB
- **DIGIRIGHTMSB** Digital audio, left channel, MSB
- **DIGIRLSB** 16-bit digital audio out (right LSB)
- **DIGIRMSB** 16-bit digital audio out (right MSB)

- **MIXREGDATA** Audio Mixer register read port
- **MIXREGSEL** Audio Mixer register select
- **PWMPDM** PWM/PDM audio encoding select
- **READBACKLSB** audio read-back LSB (source selected by \$D6F4)
- **READBACKMSB** audio read-back MSB (source selected by \$D6F4)

MISCELLANEOUS I/O FUNCTIONS

S 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	1024KB or 1,048,576 B	MB

BASE CONVERSION

Decimal	Binary	Hexadecimal
0	%0	\$0
1	%1	\$1
2	%10	\$2
3	%11	\$3
4	%100	\$4
5	%101	\$5
6	%110	\$6
7	%111	\$7
8	%1000	\$8
9	%1001	\$9
10	%1010	\$A
11	%1011	\$B
12	%1100	\$C
13	%1101	\$D
14	%1110	\$E
15	%1111	\$F
16	%10000	\$10
17	%10001	\$11
18	%10010	\$12
19	%10011	\$13
20	%10100	\$14
21	%10101	\$15
22	%10110	\$16
23	%10111	\$17
24	%11000	\$18
25	%11001	\$19
26	%11010	\$1A
27	%11011	\$1B
28	%11100	\$1C
29	%11101	\$1D
30	%11110	\$1E
31	%11111	\$1F

Decimal	Binary	Hexadecimal
32	%100000	\$20
33	%100001	\$21
34	%100010	\$22
35	%100011	\$23
36	%100100	\$24
37	%100101	\$25
38	%100110	\$26
39	%100111	\$27
40	%101000	\$28
41	%101001	\$29
42	%101010	\$2A
43	%101011	\$2B
44	%101100	\$2C
45	%101101	\$2D
46	%101110	\$2E
47	%101111	\$2F
48	%110000	\$30
49	%110001	\$31
50	%110010	\$32
51	%110011	\$33
52	%110100	\$34
53	%110101	\$35
54	%110110	\$36
55	%110111	\$37
56	%111000	\$38
57	%111001	\$39
58	%111010	\$3A
59	%111011	\$3B
60	%111100	\$3C
61	%111101	\$3D
62	%111110	\$3E
63	%111111	\$3F

Decimal	Binary	Hexadecimal
64	%1000000	\$40
65	%1000001	\$41
66	%1000010	\$42
67	%1000011	\$43
68	%1000100	\$44
69	%1000101	\$45
70	%1000110	\$46
71	%1000111	\$47
72	%1001000	\$48
73	%1001001	\$49
74	%1001010	\$4A
75	%1001011	\$4B
76	%1001100	\$4C
77	%1001101	\$4D
78	%1001110	\$4E
79	%1001111	\$4F
80	%1010000	\$50
81	%1010001	\$51
82	%1010010	\$52
83	%1010011	\$53
84	%1010100	\$54
85	%1010101	\$55
86	%1010110	\$56
87	%1010111	\$57
88	%1011000	\$58
89	%1011001	\$59
90	%1011010	\$5A
91	%1011011	\$5B
92	%1011100	\$5C
93	%1011101	\$5D
94	%1011110	\$5E
95	%1011111	\$5F

Decimal	Binary	Hexadecimal
96	%1100000	\$60
97	%1100001	\$61
98	%1100010	\$62
99	%1100011	\$63
100	%1100100	\$64
101	%1100101	\$65
102	%1100110	\$66
103	%1100111	\$67
104	%1101000	\$68
105	%1101001	\$69
106	%1101010	\$6A
107	%1101011	\$6B
108	%1101100	\$6C
109	%1101101	\$6D
110	%1101110	\$6E
111	%1101111	\$6F
112	%1110000	\$70
113	%1110001	\$71
114	%1110010	\$72
115	%1110011	\$73
116	%1110100	\$74
117	%1110101	\$75
118	%1110110	\$76
119	%1110111	\$77
120	%1111000	\$78
121	%1111001	\$79
122	%1111010	\$7A
123	%1111011	\$7B
124	%1111100	\$7C
125	%1111101	\$7D
126	%1111110	\$7E
127	%1111111	\$7F

Decimal	Binary	Hexadecimal
128	%10000000	\$80
129	%10000001	\$81
130	%10000010	\$82
131	%10000011	\$83
132	%10000100	\$84
133	%10000101	\$85
134	%10000110	\$86
135	%10000111	\$87
136	%10001000	\$88
137	%10001001	\$89
138	%10001010	\$8A
139	%10001011	\$8B
140	%10001100	\$8C
141	%10001101	\$8D
142	%10001110	\$8E
143	%10001111	\$8F
144	%10010000	\$90
145	%10010001	\$91
146	%10010010	\$92
147	%10010011	\$93
148	%10010100	\$94
149	%10010101	\$95
150	%10010110	\$96
151	%10010111	\$97
152	%10011000	\$98
153	%10011001	\$99
154	%10011010	\$9A
155	%10011011	\$9B
156	%10011100	\$9C
157	%10011101	\$9D
158	%10011110	\$9E
159	%10011111	\$9F

Decimal	Binary	Hexadecimal
160	%10100000	\$A0
161	%10100001	\$A1
162	%10100010	\$A2
163	%10100011	\$A3
164	%10100100	\$A4
165	%10100101	\$A5
166	%10100110	\$A6
167	%10100111	\$A7
168	%10101000	\$A8
169	%10101001	\$A9
170	%10101010	\$AA
171	%10101011	\$AB
172	%10101100	\$AC
173	%10101101	\$AD
174	%10101110	\$AE
175	%10101111	\$AF
176	%10110000	\$B0
177	%10110001	\$B1
178	%10110010	\$B2
179	%10110011	\$B3
180	%10110100	\$B4
181	%10110101	\$B5
182	%10110110	\$B6
183	%10110111	\$B7
184	%10111000	\$B8
185	%10111001	\$B9
186	%10111010	\$BA
187	%10111011	\$BB
188	%10111100	\$BC
189	%10111101	\$BD
190	%10111110	\$BE
191	%10111111	\$BF

Decimal	Binary	Hexadecimal
192	%11000000	\$C0
193	%11000001	\$C1
194	%11000010	\$C2
195	%11000011	\$C3
196	%11000100	\$C4
197	%11000101	\$C5
198	%11000110	\$C6
199	%11000111	\$C7
200	%11001000	\$C8
201	%11001001	\$C9
202	%11001010	\$CA
203	%11001011	\$CB
204	%11001100	\$CC
205	%11001101	\$CD
206	%11001110	\$CE
207	%11001111	\$CF
208	%11010000	\$D0
209	%11010001	\$D1
210	%11010010	\$D2
211	%11010011	\$D3
212	%11010100	\$D4
213	%11010101	\$D5
214	%11010110	\$D6
215	%11010111	\$D7
216	%11011000	\$D8
217	%11011001	\$D9
218	%11011010	\$DA
219	%11011011	\$DB
220	%11011100	\$DC
221	%11011101	\$DD
222	%11011110	\$DE
223	%11011111	\$DF

Decimal	Binary	Hexadecimal
224	%11100000	\$E0
225	%11100001	\$E1
226	%11100010	\$E2
227	%11100011	\$E3
228	%11100100	\$E4
229	%11100101	\$E5
230	%11100110	\$E6
231	%11100111	\$E7
232	%11101000	\$E8
233	%11101001	\$E9
234	%11101010	\$EA
235	%11101011	\$EB
236	%11101100	\$EC
237	%11101101	\$ED
238	%11101110	\$EE
239	%11101111	\$EF
240	%11110000	\$F0
241	%11110001	\$F1
242	%11110010	\$F2
243	%11110011	\$F3
244	%11110100	\$F4
245	%11110101	\$F5
246	%11110110	\$F6
247	%11110111	\$F7
248	%11111000	\$F8
249	%11111001	\$F9
250	%11111010	\$FA
251	%11111011	\$FB
252	%11111100	\$FC
253	%11111101	\$FD
254	%11111110	\$FE
255	%11111111	\$FF

T

APPENDIX

Flashing the FPGAs and CPLDs in the MEGA65

- Suggested PC specifications
- Warning
- Installing Vivado
- Installing the FTDI drivers
- Flashing the main FPGA using 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 also to 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 probably want to look in Chapter/Appendix [5 on page 5-5](#) instead.

This chapter is more intended for people who want to help develop cores for the MEGA65. This chapter may also be of interest to Nexys4 board owners that are interested booting their devices from the on-board QSPI flash memory chip (rather than a bitstream file on the SD card). This will require flashing an .mcs file onto their board's QSPI chip, so as to provide an initial bitstream in the 'Slot 0' position.

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 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, exciting things can be created with them.

SUGGESTED PC SPECIFICATIONS

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 12GB RAM (preferably 16GB) if you want to write programs for FPGAs and CPLDs. On the other hand, if 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.

- OS: Linux or Windows
- CPU Speed: As fast as you can get your hands on!
- Number of cores: Ideally, 8 or more, as the free license of Vivado can make use of a max of 8 cores.
- Hard disk space: Have about 70GB or more. The exact amount used depends on how many components within Vivado you install (bear in mind that the full install file is about 50GB in itself)

- Memory: minimum of 12GB (ideally, have more, to play it safe)

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, we cannot reasonably honour the warranty of the device as a computer. You have been warned!

INSTALLING VIVADO

Installation of Vivado is required to flash the QSPI flash memory within your MEGA65 target device, whether it be a MEGA65 R2/R3/R3A/R4, Nexys4/Nexys4DDR/NexysA7, MEGAphone or other.

Vivado is also the tool used to perform compilation (synthesis, as it is preferably called) of FPGA bitstreams.

To get started, connect to <https://www.xilinx.com/support/download.html>

Select 2020.2 version

The screenshot shows the Xilinx Vivado Design Suite - HLx Editions - 2020.2 Full Product Installation page. On the left, a sidebar lists versions: 2020.3, 2020.2 (highlighted with a yellow box), 2020.1, 2019.2, Vivado Archive, ISE Archive, and CAE Vendor Libraries Archive. The main content area displays the following information:

Vivado Design Suite - HLx Editions - 2020.2 Full Product Installation

Important Information

Vivado® Design Suite 2020.2 is now available

- Public access support for the Xilinx® Versal™ Platforms
- Petalinux now a part of Xilinx Unified Installer
- Access Block Design container now to create team-based designs
- Abstract Shell for Dynamic Function eXchange
- 2020.2 Introduces Vitis™ HLS for Vivado flows
- Add-on for MATLAB® and Simulink® (Unified Model Composer and System Generator)

We strongly recommend to use the web installers as it reduces download time and saves significant disk space.

On the right side, there are four columns with corresponding links:

Download Includes	Download Type	Last Updated	Answers
			Documentation
			Support Forums

NOTE : Some users still have success with using older versions, as the main aim here is to install a version that supports the FPGA of your target hardware.

i.e., the Artix7 100T (for Nexys and R2) or 200T (R3/R3A/R4).

Click on Xilinx Unified Installer 2020.2: Windows Self Extracting Web Installer EXE - 248.44MB

The screenshot shows the Xilinx Unified Installer 2020.2: Windows Self Extracting Web Installer (EXE - 248.44 MB) download verification page. It includes the MD5 SUM Value and three download verification options: Digests, Signature, and Public Key.

Xilinx Unified Installer 2020.2: Windows Self Extracting Web Installer (EXE - 248.44 MB)

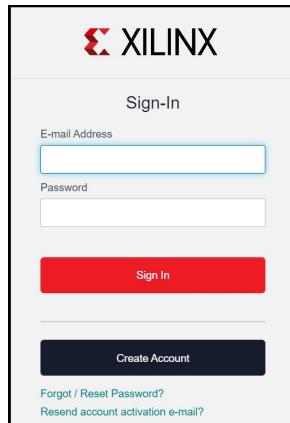
MD5 SUM Value : 102bb67c6806a6667dc7176be7997475

Download Verification i

Digests Signature Public Key

You will be asked to create an account in order to sign in and be able to download the installation program.

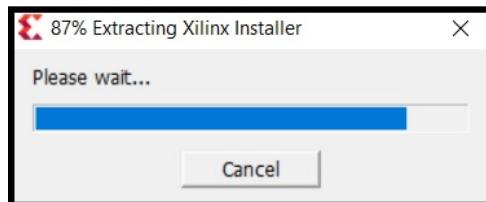
Your credentials will also be requested when doing the installation.



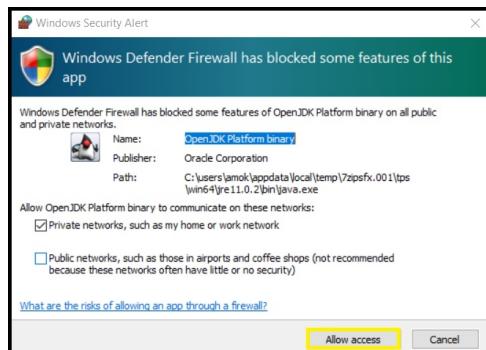
After having signed in, you have to provide some personal information and then click on Download

A screenshot of a Xilinx personal information form. The form includes fields for "Business E-mail*", "Company Name*", "Address 1*", "Address 2", "Location*", "City*", "Phone", "Job Function*", and "State/Province" and "Postal Code". Most fields are highlighted with yellow. A note at the bottom states: "For more information about how we process your personal information, please see our [privacy policy](#)". A red "Download" button is located at the bottom left.

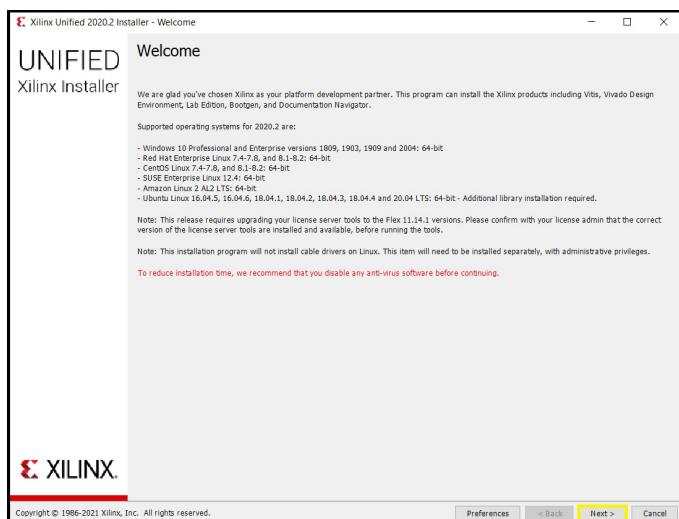
Execute the installer as Administrator (Xilinx_Unified_2020.2_1118_1232_Win64.exe).



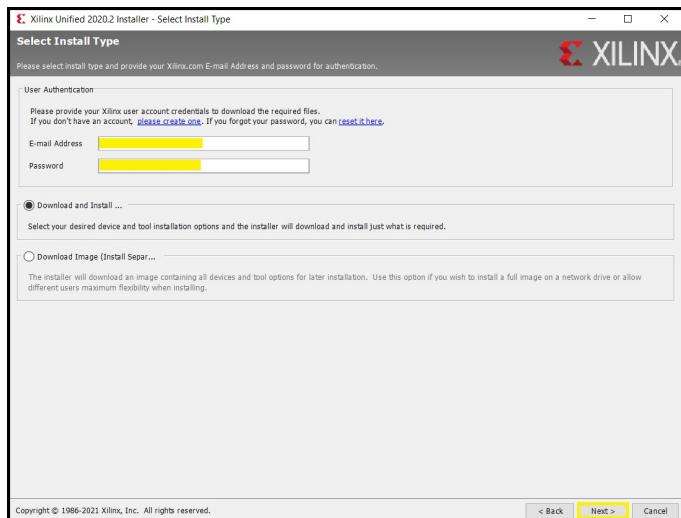
Click on Allow Access.



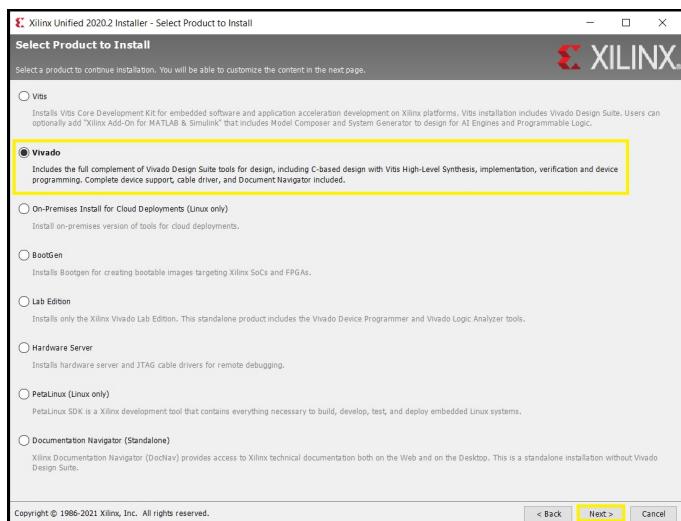
Click on Next.



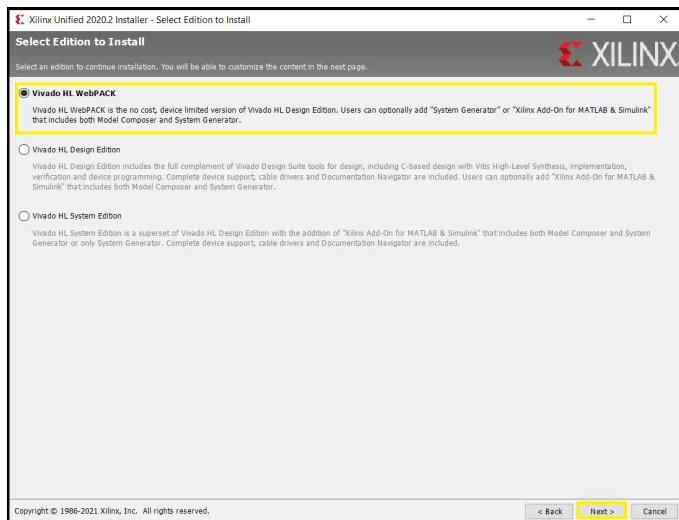
Enter your credentials and click on Next.



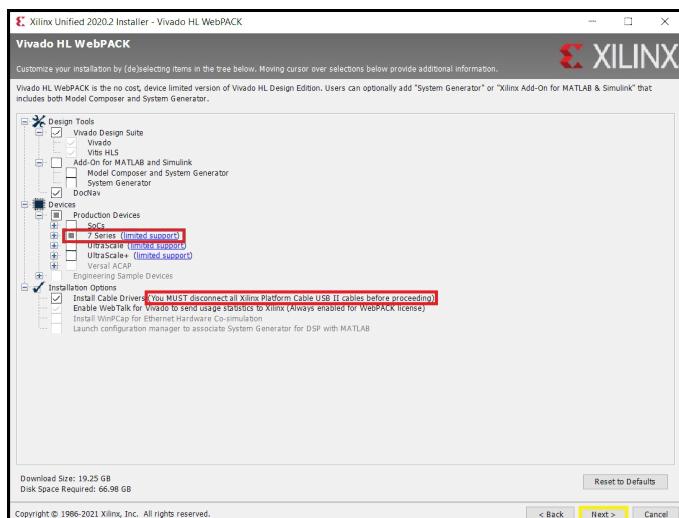
Select Vivado and click on Next.



Select "Vivado HL WebPACK" and click on "Next"

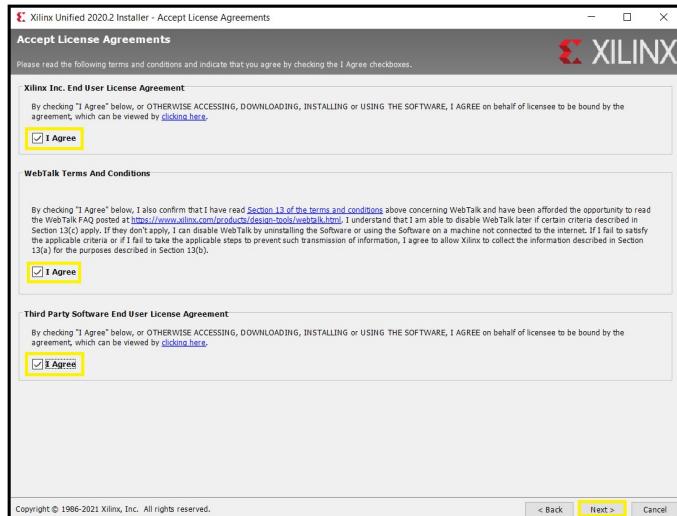


We'd suggest selecting only the **"7 Series"** devices, as our chosen FPGA is within this series, and de-selecting the other series will save you about 6GB in download size. Then click on "Next"

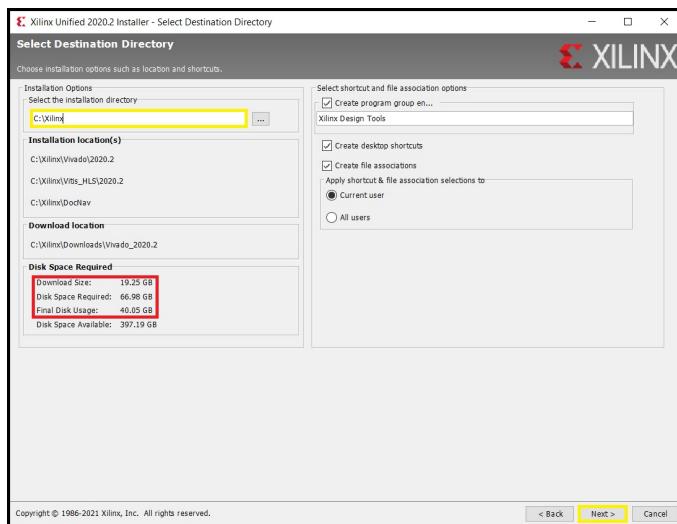


Warning: As stated, disconnect any USB cable that would be connected to your PC from the Nexys board.

Agree with all the End User Licence Agreement and Terms and conditions and click on "Next".

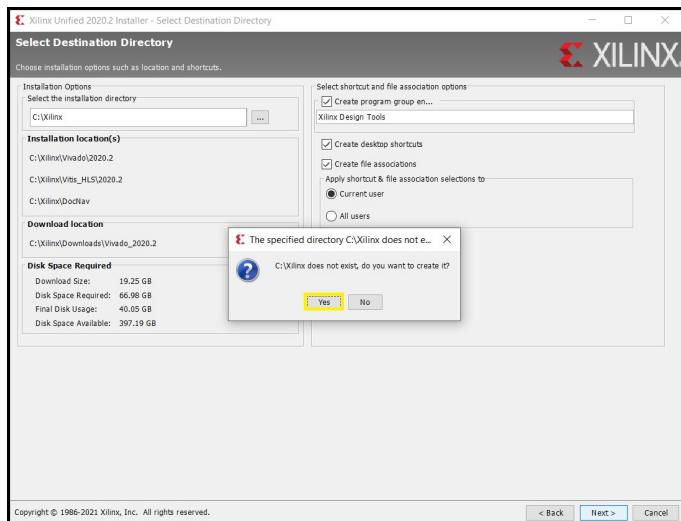


Choose the location where you want to install the software and click on "Next".

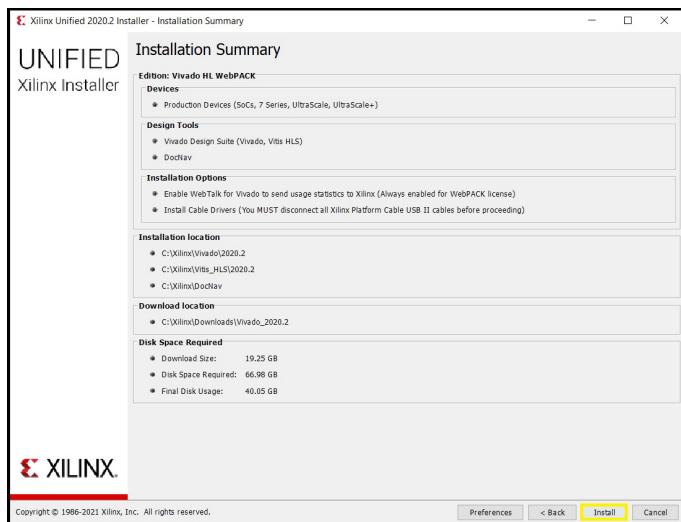


Warning : You are about to download 20GB of software and you need 70GB to perform the installation.

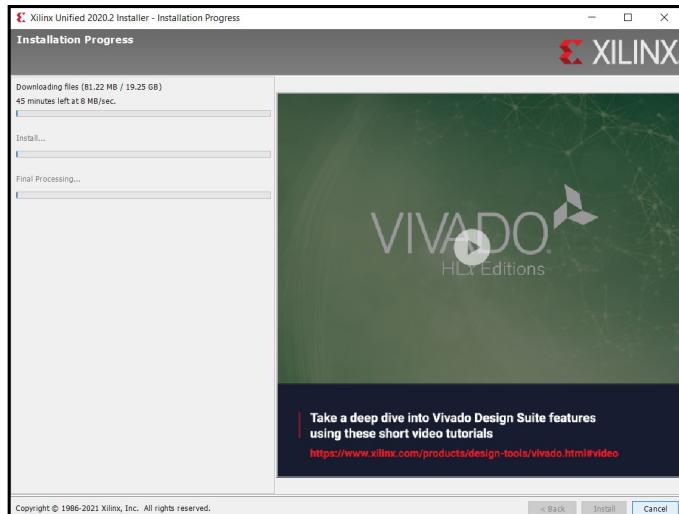
Click on "Yes"



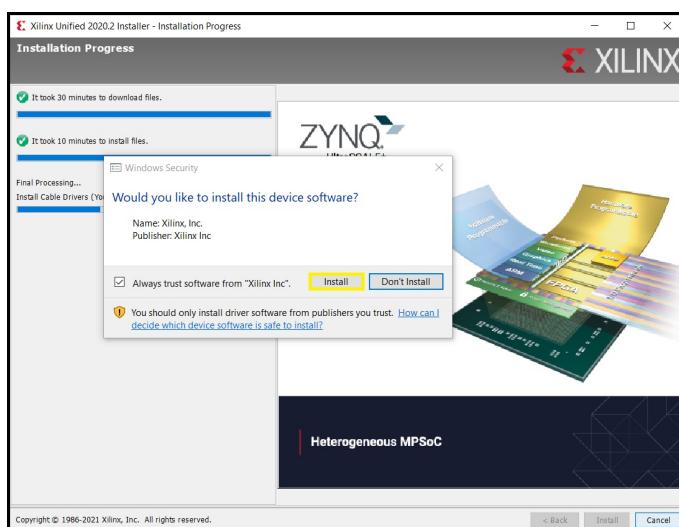
Click on "Install"



Wait for the installation to complete. At the very end of the installation you will be asked if you want to install Xilinx device software.

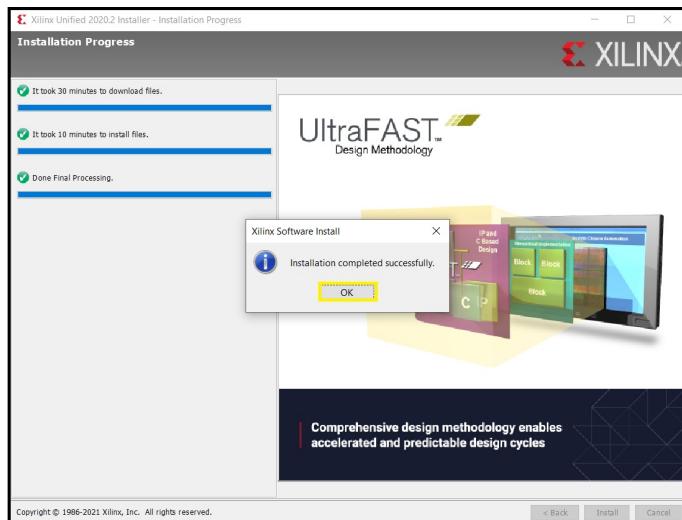


Click on "Install"



Let the installation complete.

The installation is completed. Click on "OK"

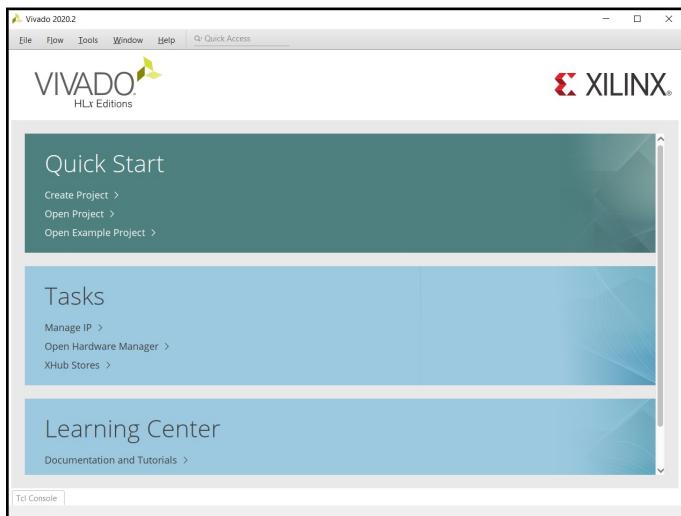


You end up with the following icons on your desktop:

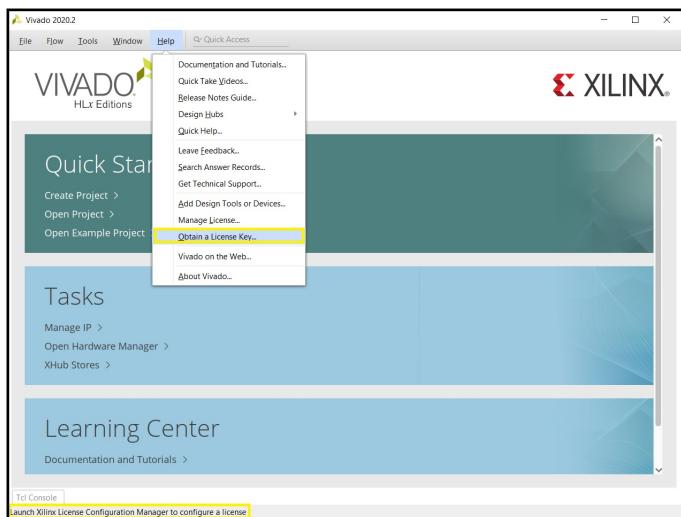


Note: Ubuntu users might need to install one missing dependency with: `sudo apt install libtinfo5`

Launch Vivado 2020.2



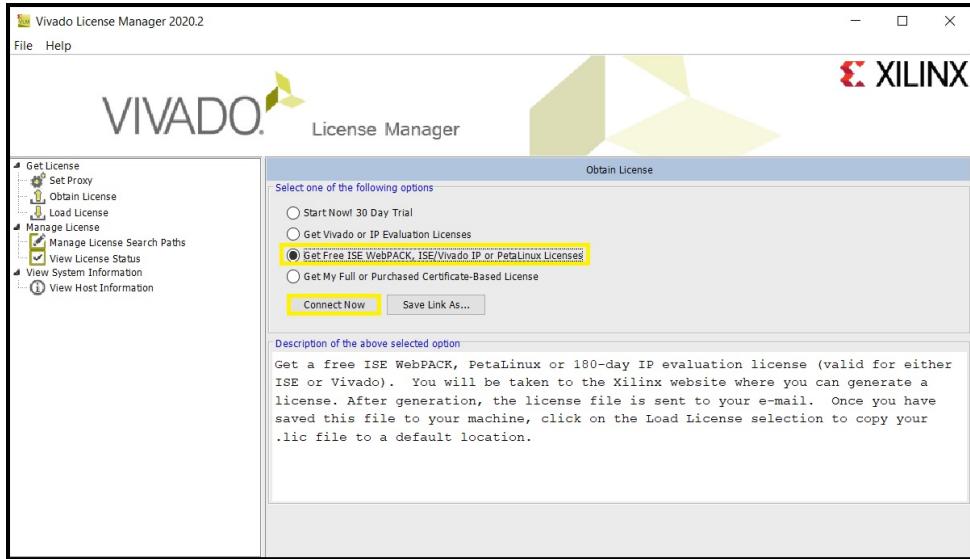
Click on "Help" ->"Obtain a licence Key"



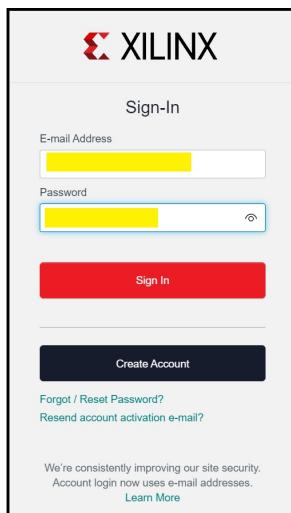
This launches the Vivado licence manager

Select "Get Free ISE WebPACK, ISE/Vivado IP or PetaLinux Licenses"

Click on "Connect Now"



Connect with the user account you have created to be able to download the Vivado software. If you were not already connected to Xilinx website, this will take you to the main webpage. Go back in the licence manager (which is not closed)



Click again on "Connect Now" (ensure "Get Free ISE WebPACK, ISE/Vivado IP or PetaLinux Licenses" is still selected)



You then register your personal information on the Vivado website, and click on "Next".

Product Licensing - Name and Address Verification

U.S. Government Export Approval

- U.S. export regulations require that your First Name, Last Name, Company Name and Shipping Address be verified before Xilinx can fulfill your download request. [Please provide accurate and complete information.](#)
- Addresses with Post Office Boxes and names/addresses with Non-Roman Characters with accents such as grave, tilde or colon [are not supported](#) by US export compliance systems.

First Name* Last Name*

Business E-mail*

Company Name*
Please enter the name of your business or institution.

Address 1*
Please enter your Company Address.

Address 2

Location* State/Province

City* Postal Code

Phone

Job Function*
Hardware Enthusiast/Pro-Hobbyist

For more information about how we process your personal information, please see our [privacy policy](#).

[Next](#)

Select "ISE WebPACK Licence" and "Vivado Design Suite: HL WebPACK 2015 and Earlier License"

Then click on "Generate Node-Locked Licence"

Create a New License File

Create a new license file by making your product selections from the table below. [?](#)

Certificate Based Licenses

Product	Type	License	Available Seats	Status	Subscription End Date
Xilinx add-on for Matlab and Simulink, 90 Day Evaluation	Certificate - Evaluation	Node	1/1	Current	90 days
ISE Embedded Edition License	Certificate - No Charge	Node	1/1	Current	None
Vivado Design Suite, 30-Day Evaluation License	Certificate - Evaluation	Node	1/1	Current	30 days
SDSoC Environment, 60 Day Evaluation License	Certificate - Evaluation	Node	1/1	Current	60 days
SDAccel OpenCL Development Environment, 30 Day Node Locked Evaluation Lice...	Certificate - Evaluation	Node	1/1	Current	30 days
<input checked="" type="checkbox"/> Vivado Design Suite: HL WebPACK 2015 and Earlier License	Certificate - No Charge	Node	1/1	Current	None
<input checked="" type="checkbox"/> ISE WebPACK License	Certificate - No Charge	Node	1/1	Current	None
Xilinx MicroBlaze/Alti Programmatic SoC Software Development Kit – Standalone	Certificate - No Charge	Node	1/1	Current	None
PetaLinux Tools License	Certificate - Evaluation	Node	1/1	Current	365 days
Vivado HLS Evaluation License	Certificate - Evaluation	Node	1/1	Current	30 days

[Generate Node-Locked License](#)

Click on "Next"

Generate Node License
Fields marked with an asterisk * are required.

1 PRODUCT SELECTION

Product Selections	Product	Type	Available Seats	Subscription End Date	Requested Seats	Borrowed Seats
*	ISE WebPACK License	No Charge	1/1	None	1	

2 SYSTEM INFORMATION

License	Node
Host ID *	Any

3 COMMENTS

Comments ?	
------------	--

Next **Cancel**

Click on "Next"

Generate Node License

4 REVIEW LICENSE REQUEST

Product Selections	Product	Subscription End Date	Available Seats	Requested Seats	
	ISE WebPACK License		1/1	1	

System Information

License	Node
Host ID	ANY

Note: WebTalk is always enabled for WebPACK users. WebTalk ignores user and install preference when a bitstream is generated using the WebPACK license. If a design is using a device contained in WebPACK and a WebPACK license is available, the WebPACK license will always be used. To get additional information on WebTalk, go to www.xilinx.com/webtalk.

Previous **Next** **Cancel**

Check your email box : You should have received an email from Xilinx, Inc. with a licence file attached and named "Xilinc.lic".

Retrieve this file on your PC and keep it in safe place.

Congratulations

Your new license file has been successfully generated and e-mailed to [REDACTED] You can also view the license file under the Manage Licenses tab.

Please add this sender (xilinx.notification@entitlenow.com) to your address book.

License File Details

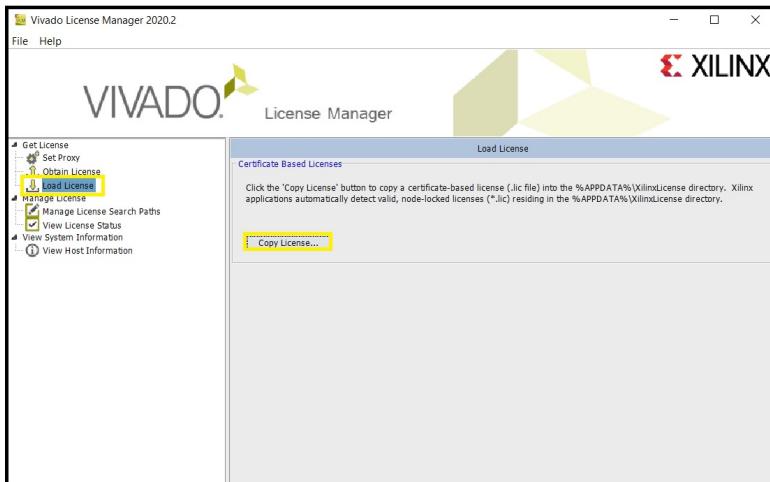
Node License
Host ID: ANY

Products

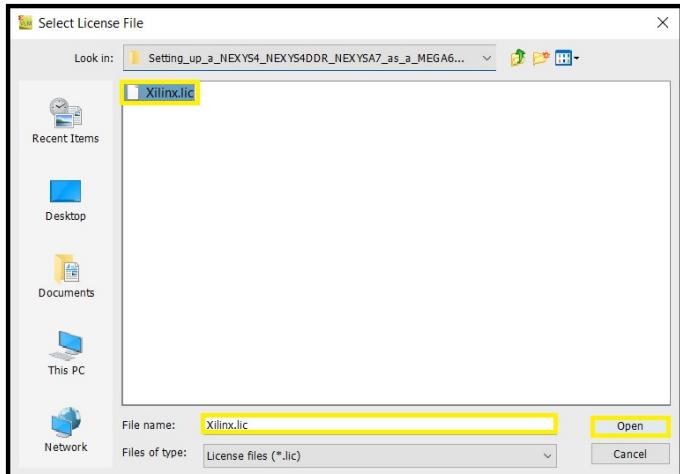
ISE WebPACK License (No Charge): 1 seats

Go back to the licence manager (which is still running).

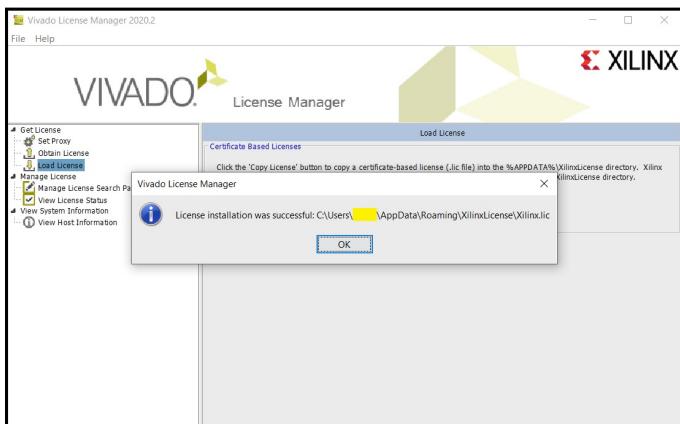
Set "Load License" and click on "Copy License"



Browse to the location where you saved "Xilinx.lic" file, select it and click on "Open".



Click on "OK" and close the Vivado licence manager.



Your Vivado software is registered and you can now use it.

INSTALLING THE FTDI DRIVERS

The FTDI drivers are needed in order for your PC to communicate with the hardware's JTAG port and serial comms port (note that the single physical USB connection made to your PC actually provides these two ports).

Linux drivers

Some Linux users have reported that they have found the FTDI drivers to be installed within their Linux distributions out-of-the-box, while others have found they needed to run this extra command after installing Vivado:

```
cd /opt/Xilinx/Vivado/2018.3/data/xicom/cable_drivers/lin64/install_script/install_driver  
sudo ./install_drivers
```

Windows drivers

Download the following archive to install the drivers:

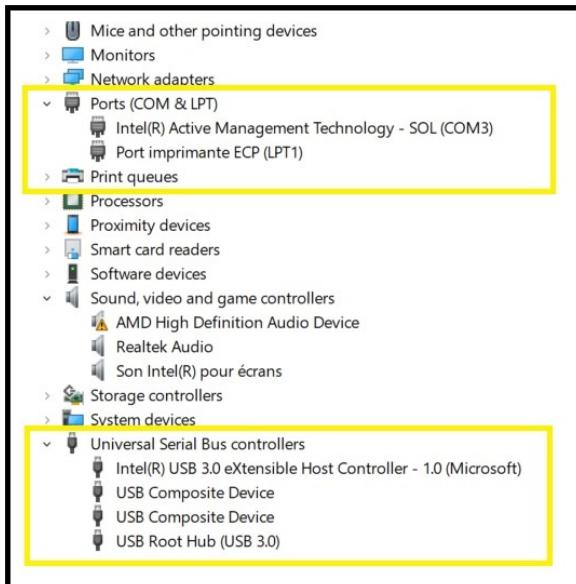
- https://www.ftdichip.com/Drivers/CDM/CDM21228_Setup.zip

Unzip the file **CDM21228_Setup.zip**, you get the file **CDM21228_Setup.exe**.

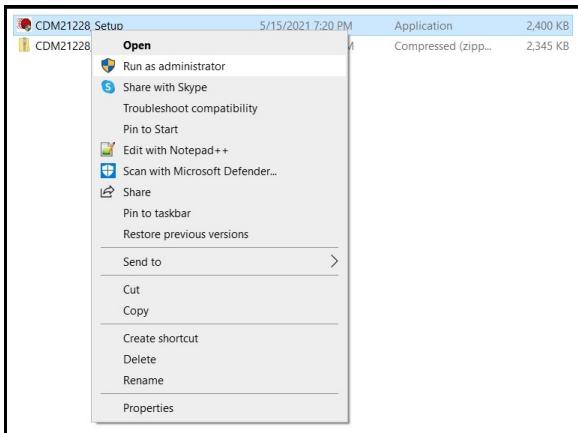
Warning:

Before installing the drivers, it is imperative to switch off the Nexys4 board and to disconnect the USB cable from the PC.

Review the devices already installed before the installation:

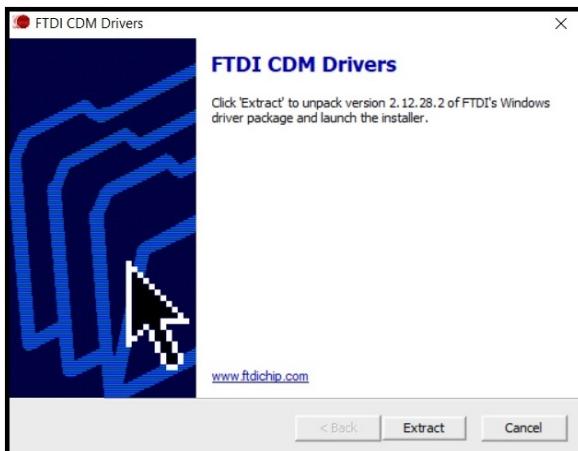


Run the file CDM21228_Setup.exe as administrator:

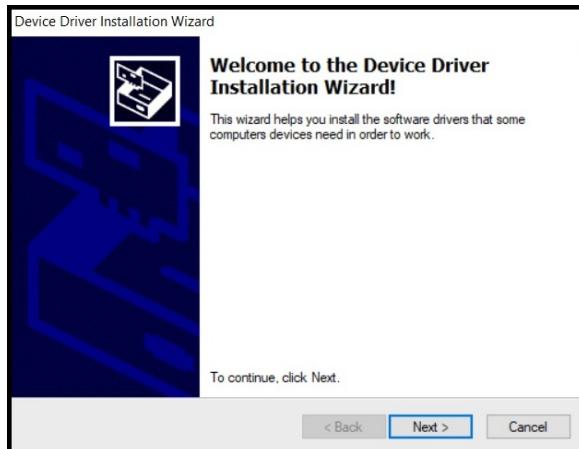


Confirm that you want to run the program.

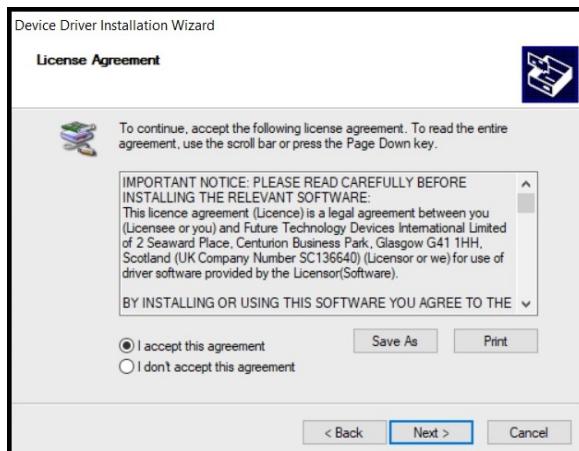
Click on "Extract".



Click on "Next >"



Accept the agreement and click on "Next >".



The installation of the drivers starts.

Click on "Finish".



Connect the USB cable to a USB port on the PC **without turning on the Nexys4 board**.

Connecting the USB cable triggers the appearance of new devices.

The screenshot compares the Windows Device Manager before and after driver installation. The left side shows the state "BEFORE DRIVERS INTALLATION" (Nexys power switch off and USB cable unplugged from PC USB port). The right side shows the state "AFTER DRIVERS INSTALLATION" (Nexys power switch off and USB cable plugged in PC USB port).

BEFORE DRIVERS INTALLATION (Nexys power switch off and USB cable unplugged from PC USB port):

- Mice and other pointing devices
- Monitors
- Network adapters
- Ports (COM & LPT)
 - Intel(R) Active Management Technology - SOL (COM3)
 - Port imprimante ECP (LPT1)
- Print queues
- Processors
- Proximity devices
- Smart card readers
- Software devices
- Sound, video and game controllers
 - AMD High Definition Audio Device
 - Realtek Audio
 - Son Intel(R) pour écrans
- Storage controllers
- System devices
- Universal Serial Bus controllers
 - Intel(R) USB 3.0 eXtensible Host Controller - 1.0 (Microsoft)
 - USB Composite Device
 - USB Composite Device
 - USB Root Hub (USB 3.0)

AFTER DRIVERS INSTALLATION (Nexys power switch off and USB cable plugged in PC USB port):

- Mice and other pointing devices
- Monitors
- Network adapters
- Ports (COM & LPT)
 - Intel(R) Active Management Technology - SOL (COM3)
 - Port imprimante ECP (LPT1)
 - USB Serial Port (COM5)
- Print queues
- Processors
- Proximity devices
- Smart card readers
- Software devices
- Sound, video and game controllers
 - AMD High Definition Audio Device
 - Realtek Audio
 - Son Intel(R) pour écrans
- Storage controllers
- System devices
- Universal Serial Bus controllers
 - Intel(R) USB 3.0 eXtensible Host Controller - 1.0 (Microsoft)
 - USB Composite Device
 - USB Composite Device
 - USB Root Hub (USB 3.0)
 - USB Composite Device
 - USB Serial Converter A
 - USB Serial Converter B

Yellow boxes highlight the newly installed COM and USB composite devices. Red boxes highlight the newly installed USB Serial Converter A and B.

- An additional COM port has been installed: **This is the COM port that will be used to communicate with the Nexys4 board.**
- An additional USB composite device has been installed.

- Two USB serial converter devices have been installed.

At this point the Nexys4 board has still not been powered up.

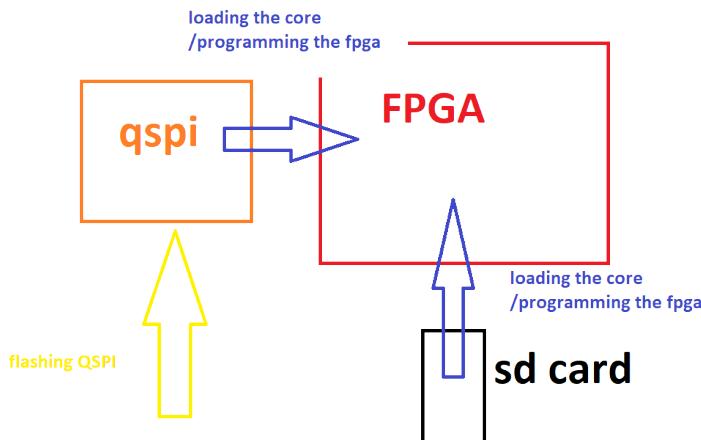
For more information about the installed drivers, you can download the corresponding documentation:

- https://ftdichip.com/wp-content/uploads/2020/08/AN_396-FTDI-Drivers-Installation-Guide-for-Windows-10.pdf
- https://ftdichip.com/wp-content/uploads/2021/01/AN_119_FTDI_Drivers_Installation_Guide_for_Windows7.pdf

FLASHING THE MAIN FPGA USING VIVADO

Firstly, to clarify that when we say 'flashing the FPGA', in reality, what we mean is that we are flashing the QSPI flash memory chip that the FPGA makes use of upon startup in order to quickly load the bitstream from.

The diagram below shows two common pathways that the FPGA can load bitstreams at startup:



- We can first flash a bitstream/core-file onto the QSPI flash memory chip, and the FPGA can load this quickly at power-up. Flashing the QSPI is quite slow, but the reward of a fast boot-up time is an advantage.

- Nexys board users can drop a bitstream file onto our SD card and let the FPGA load it (somewhat more slowly) from there at power-up. This allows them to swap/upgrade bitstreams quickly without the need for a TE0790-03 JTAG programming module. This feature is planned to be added to the MEGA65 later, though.

In this section, we describe the pathway that makes use of the QSPI.

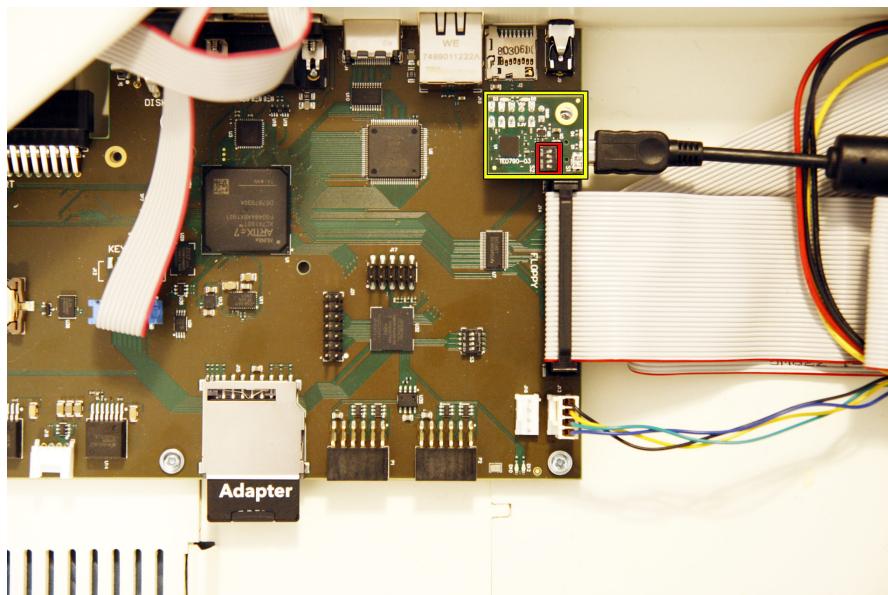
Many of the following steps in this section are applicable not only to MEGA65 R2/R3/R3A/R4 owners, but Nexys4 board owners too. There are a few points of distinction along the way that readers will be made aware of.

If you choose to proceed, you will need a functioning installation of Xilinx's Vivado software, and the FTDI drivers installed, as described in the earlier sections.

You will also need to download or build an .mcs bitstream file (and optional .prm checksum verification file) that you intend flash onto the QSPI chip via Vivado. See [Bitstream files](#) for more details on where such files can be downloaded.

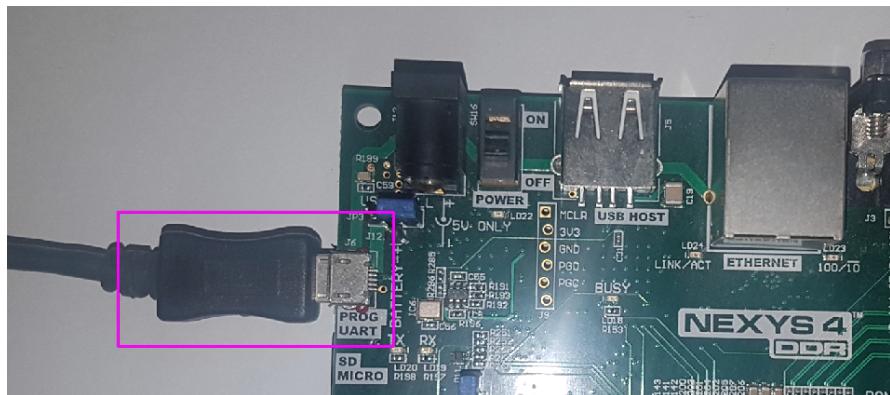
For MEGA65 R2/R3/R3A/R4 owners:

You will need a TE0790-03 JTAG programming module. 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.

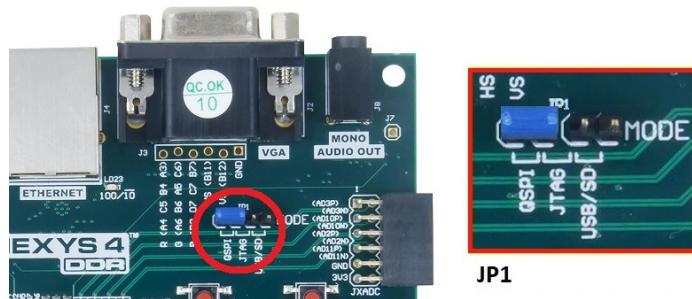


For Nexys4 board owners:

Simply connect your micro-usb cable between your Nexys4 board and your PC via the port labeled 'PROG UART' (J6), as shown:



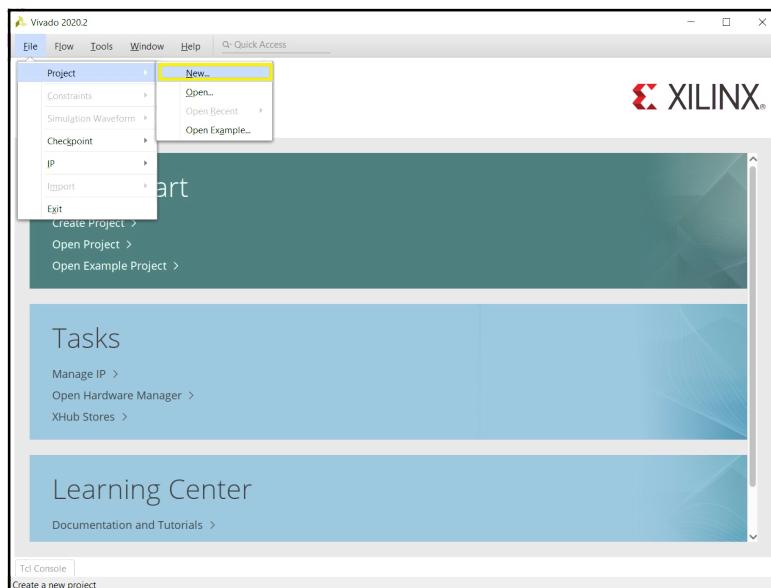
Also, set J1 jumper to the QSPI position:



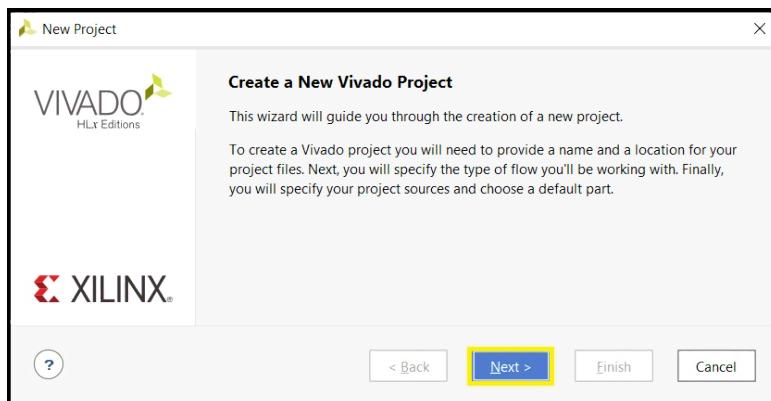
- Connect your non-8-bit computer to the FPGA programming device using the appropriate USB cable.
- Switch the MEGA65 computer ON.
- Open Vivado.

Step 1a: Create a new Vivado project with "File", "Project", "New...".

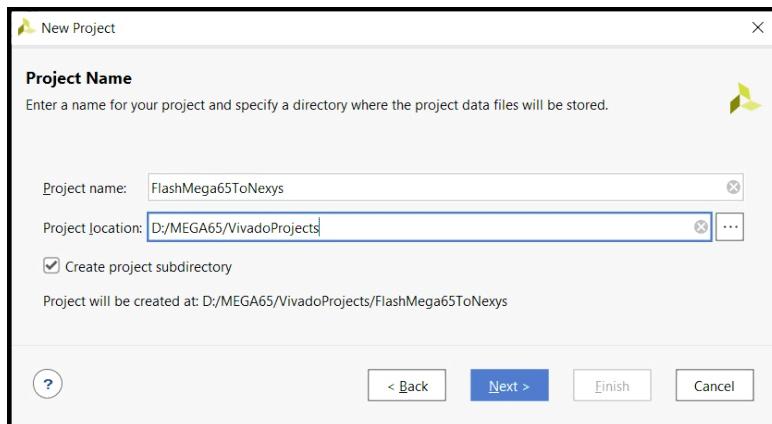
NOTE: On future occasions that you need to flash the QSPI, just re-open this project (no need to create a new project each time).



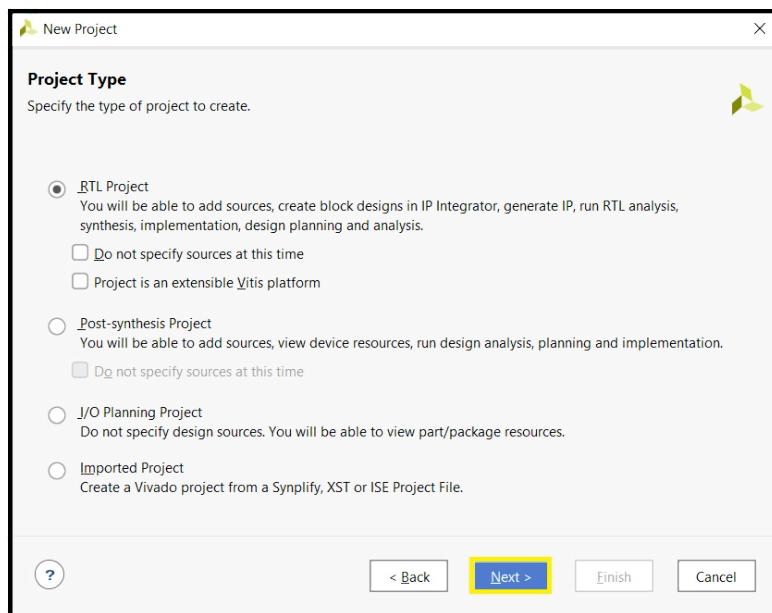
Step 1b: The 'New Project' wizard appears. Click on "Next":



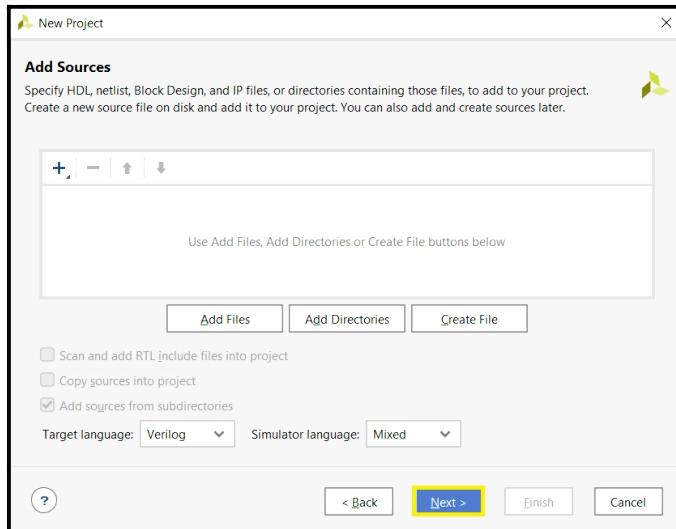
Step 1c: Name your project and choose the location you like, then click on "Next":



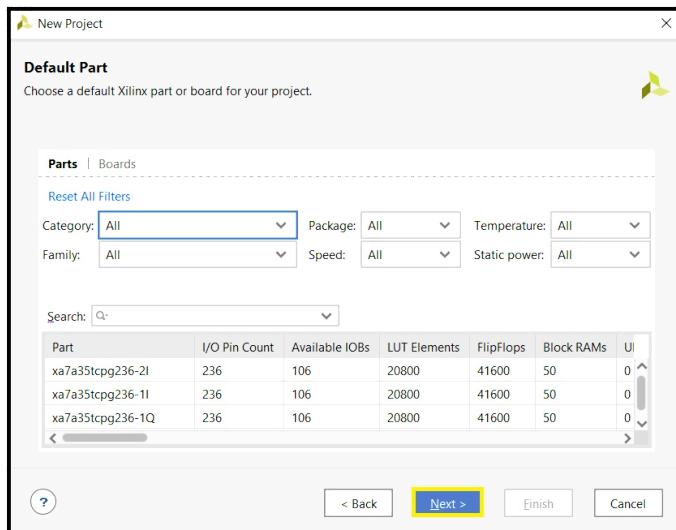
Step 1d: Keep the default selected options and click on "Next":



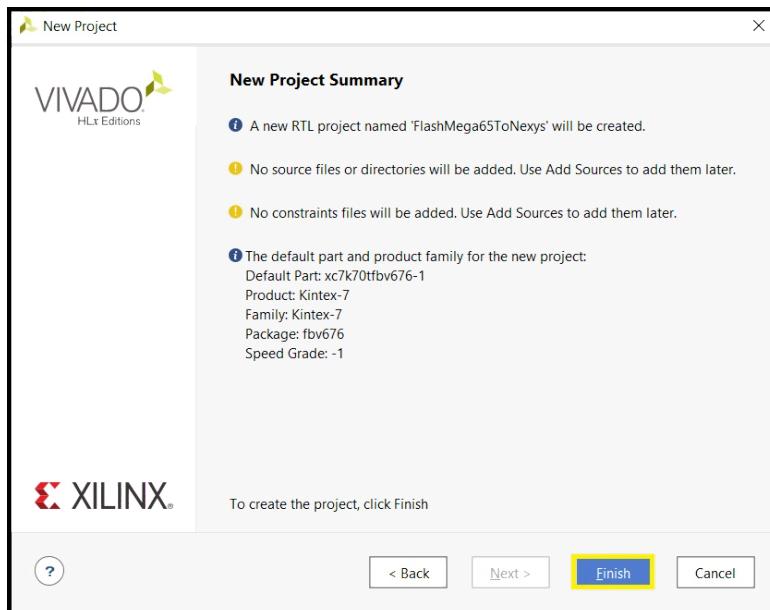
Step 1e: Do not add any sources, keep the default selected options and click on "Next":



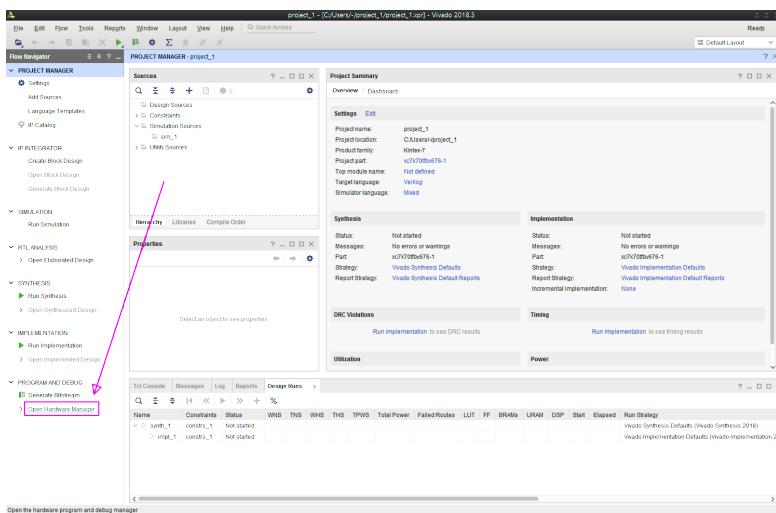
Step 1f: Keep the default selected options and click on "Next":



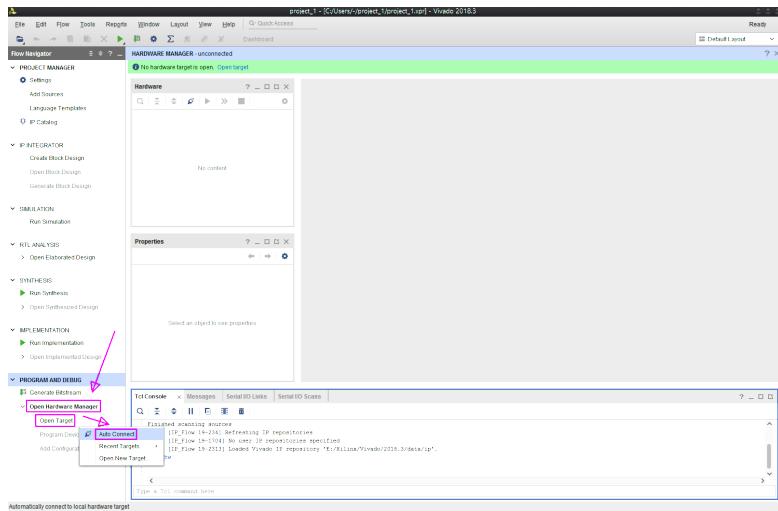
Step 1g: Click on "Finish":



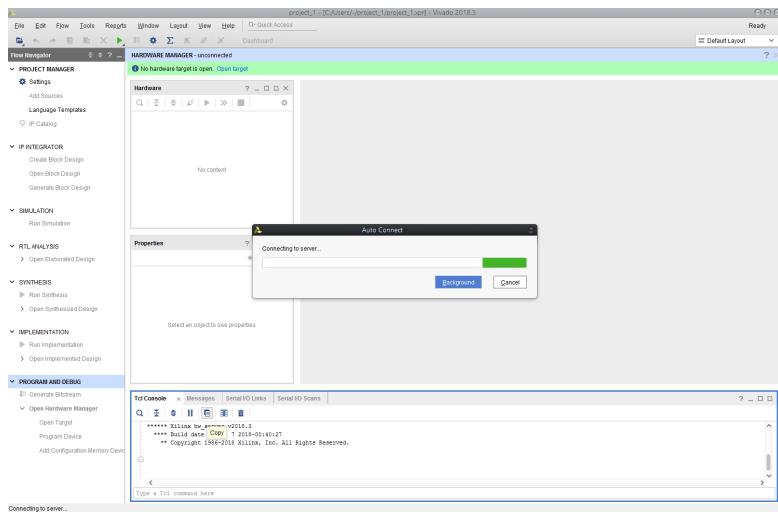
Step 2: In the left column, select "Open Hardware Manager" at the very bottom.



Step 3: Connect to the FPGA:
Under "Open Hardware Manager", choose "Open Target", then "Auto Connect".

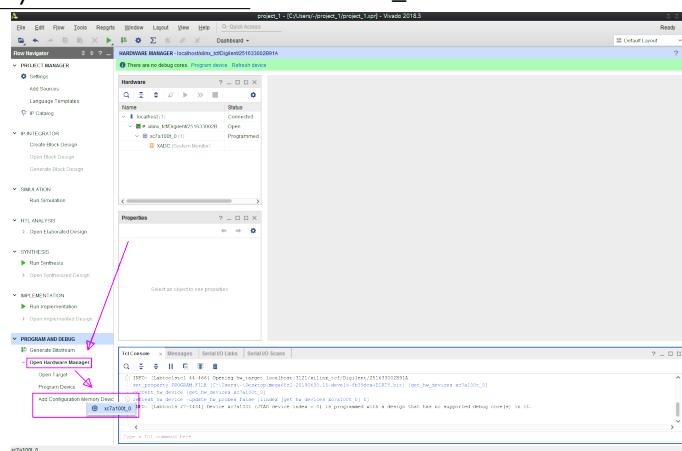


Step 4: Wait a moment, "Connecting to server..." should automatically close without dropping an error to the console.



Step 5: Under "Open Hardware Manager", choose "Add Configuration Memory Device", then:

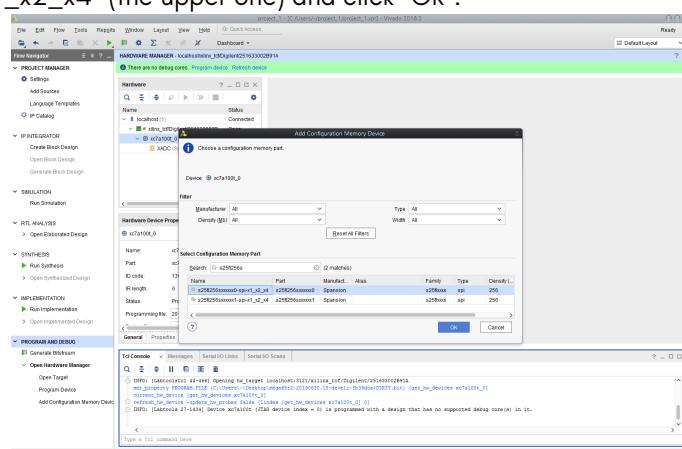
- For MEGA65R3/R3A/R4: "xc7a200t_0"
- For Nexys4 and MEGA65R2: "xc7a100t_0".



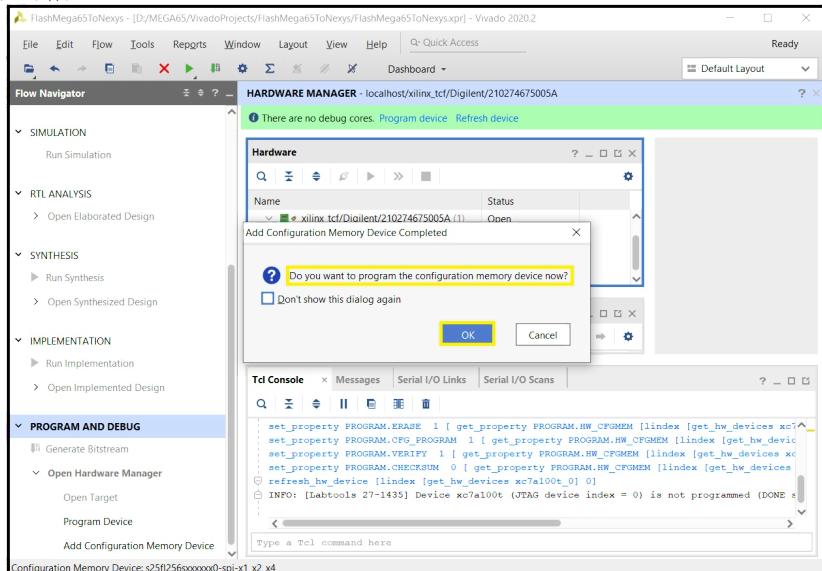
Step 6a: Select Memory Part:

In the newly opened dialogue:

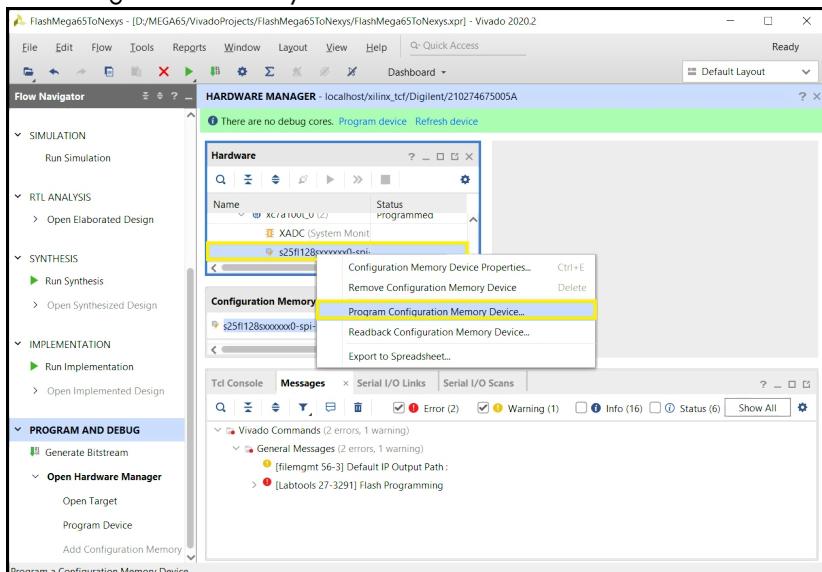
- For MEGA65R2/R3: type "S25fl256s" (without quotes), then select "s25fl256xxxxxxxxx0-spi-x1_x2_x4" (the upper one) and click "OK".
- For MEGA65R3A/R4: Vivado cannot flash the larger flash part on these boards. Use the flash menu on the MEGA65.
- For Nexys4: type "S25fl128s" (without quotes), then select "s25fl128xxxxxxxxx0-spi-x1_x2_x4" (the upper one) and click "OK".



Step 6b: Click on "OK" to confirm you want to program the configuration memory device now.

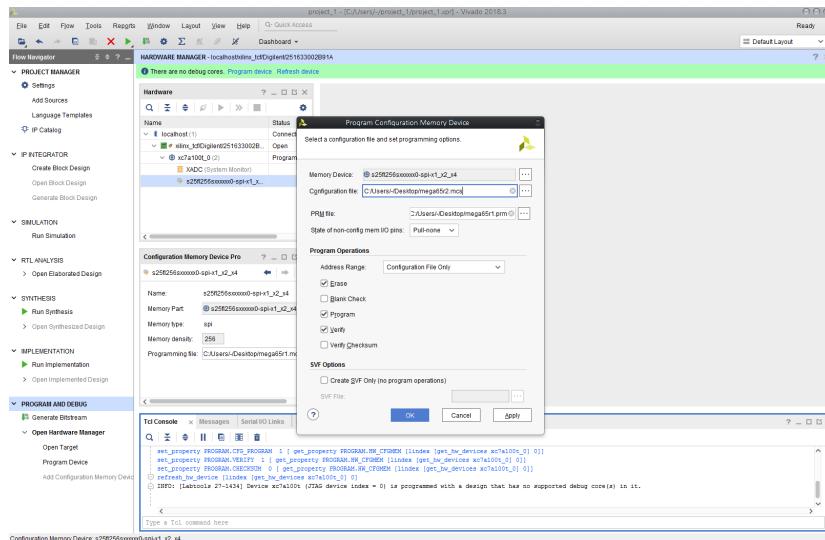


Step 6c: If you do not see such a popup, or wish to reprogram the QSPI on a future occasion, in "Hardware" window, right click on the memory configuration and select "Program Configuration Memory Device":

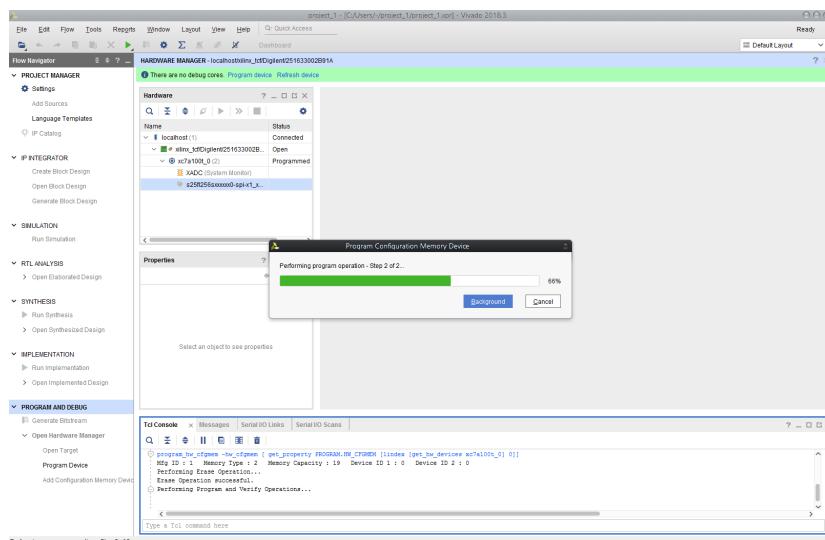


Step 7: Set programming options:

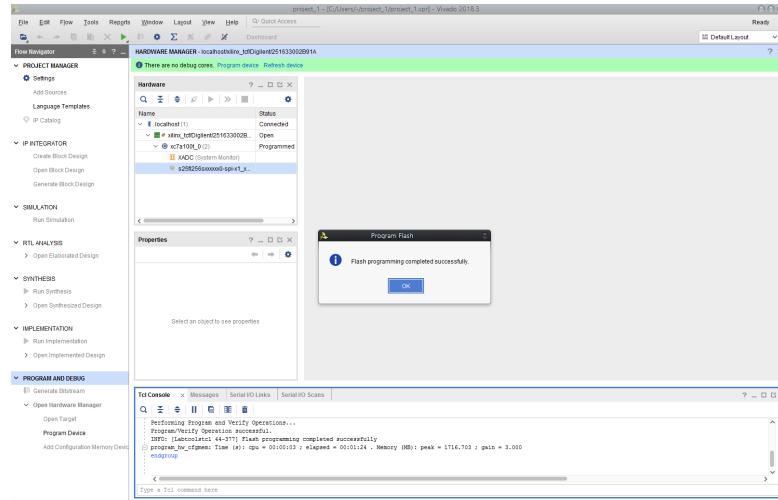
In the next dialogue, set the "Configuration file" to the path of your ".mcs" bitstream file. You can also optionally set the "PRM file" field to the path of your ".prm" file. Leave all other parameters as they are (see screenshot below).



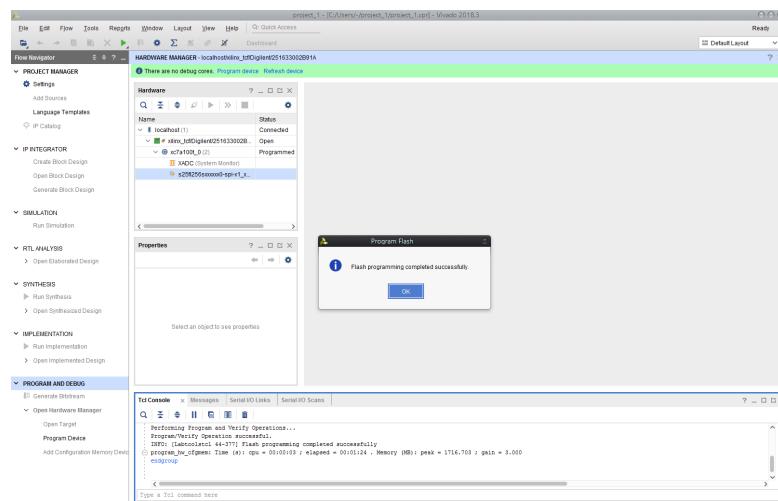
Step 8: Patiently wait for the programming to finish. This can take several minutes as the Vivado software erases and then reprograms the flash memory that is used to initialise the FPGA on power-up.



Step 9: If your screen looks like the screenshot below, your new bitstream has been successfully flashed into Slot0 of your QSPI flash memory!

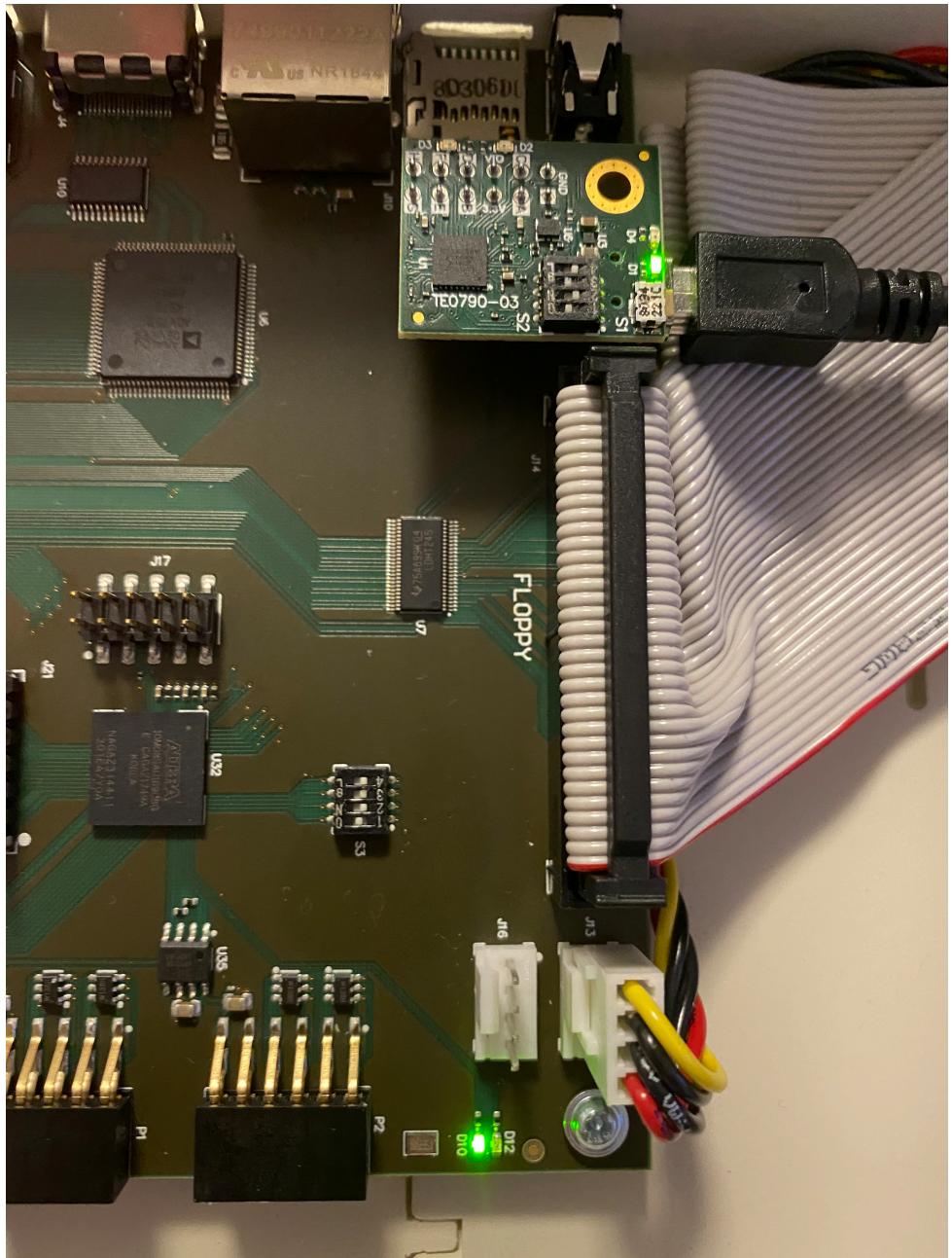


Step 10: If you want to reflash the FPGA, you might find the "Add Configuration Memory Device" option in step 5 greyed out. Instead, select "s25fl256xxxxxxxx0-spi-x1_x2_x4" in the "Hardware" window, press right mouse button and select "Program Configuration Memory Device" to flash.



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 and 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.



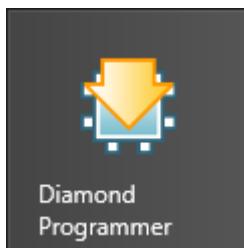
T-41

On the PCB of a R2/R3/R3A/R4 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 "ambulance mode" (blue flashing lights) 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 the Diamond Programmer which can be downloaded from the Internet.

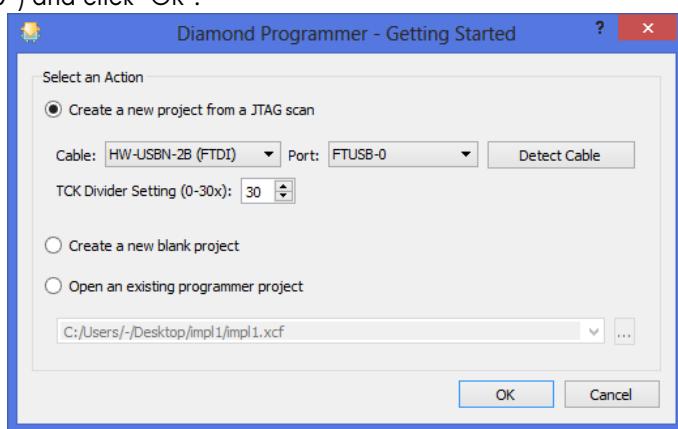
Step 1: Open DIAMOND PROGRAMMER:

Select "Create a new project from a JTAG scan". If entry under "Cable:" is empty, click "Detect Cable".



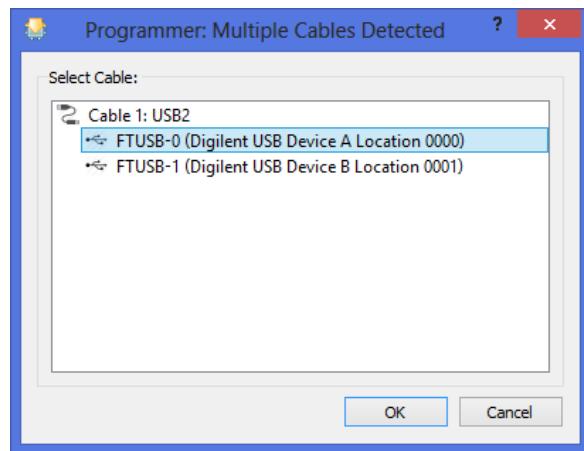
Step 2: Create a new project:

If dialog "Programmer: Multiple Cables Detected" appears, select the first entry ("Location 0000") and click "OK".



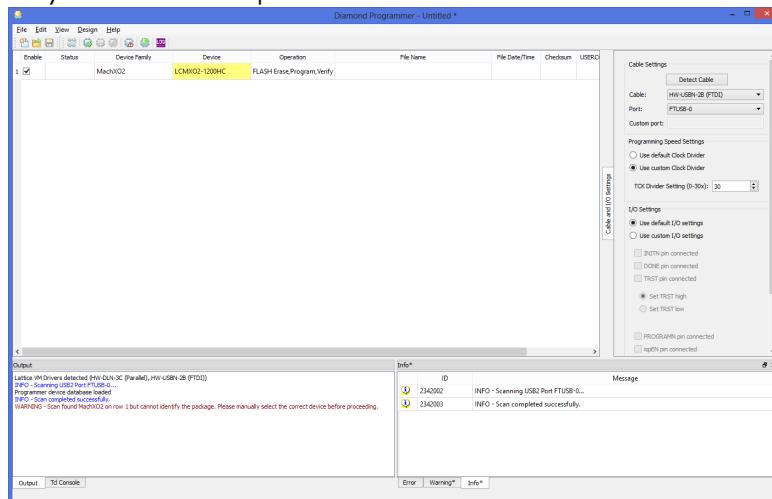
Step 3: Select cable:

You have now created a new project which should display "MachXO2" under "Device Family" and "LCMXO2-1200HC" under "Device"



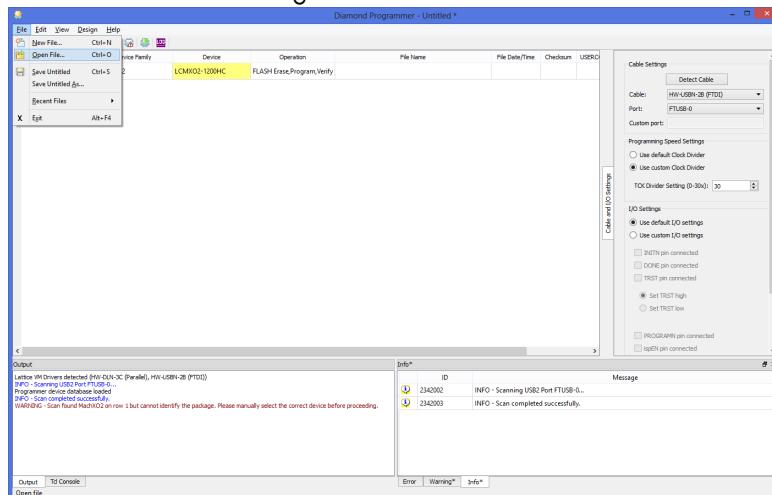
Step 4: New Diamond Programmer project:

Choose "File" then "Open File" to load the Diamond Pprogrammer project with the MEGA65 keyboard firmware update.



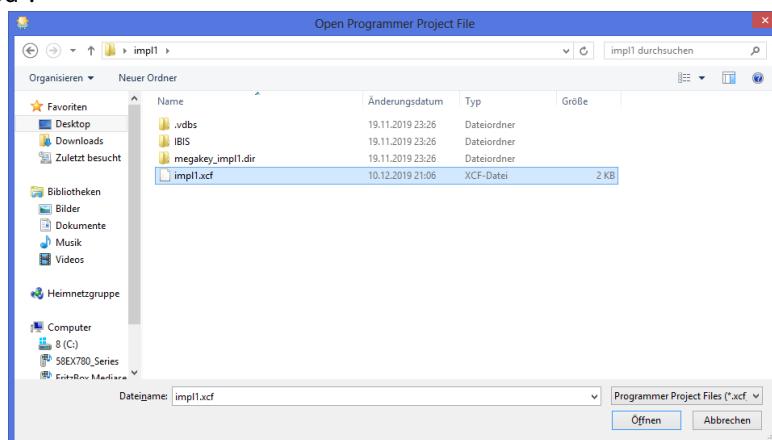
Step 5: Open project:

Navigate into the folder with the extracted MEGA65 keyboard firmware files you have received and select the file ending with ".xcf".

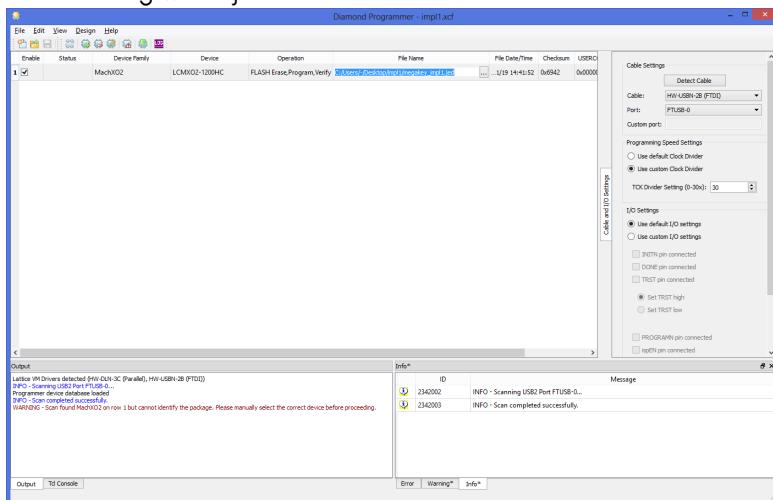


Step 6: Select project file:

Click the three dots under "File Name" to set the correct path and find the file ending with ".jed".

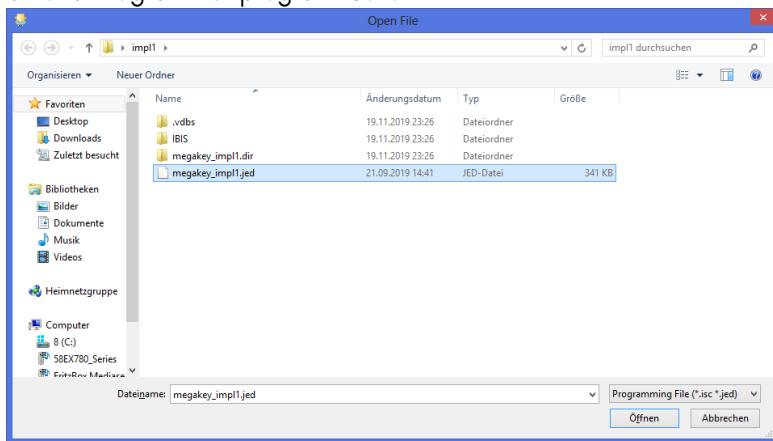


Step 7: Choose correct path of .jed file:
Select the file ending with ".jed" and click "OK".



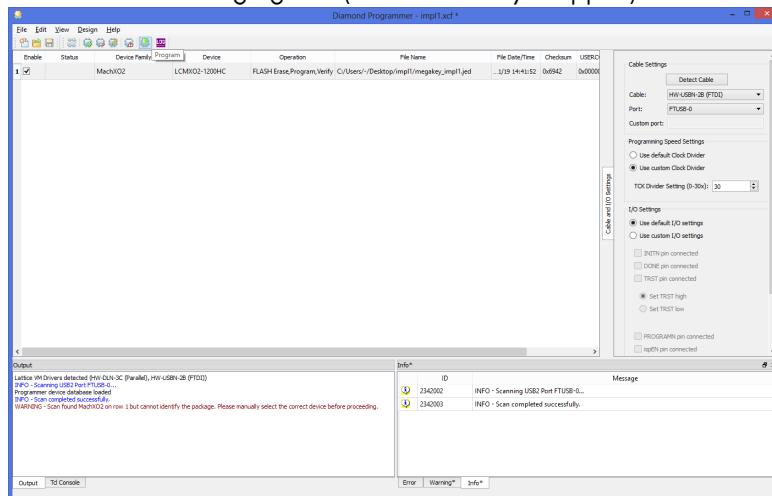
Step 8: Select .jed file:

Click on the icon with the green arrow facing down "PROGRAM", which looks similar to the Diamond Programmer program icon.



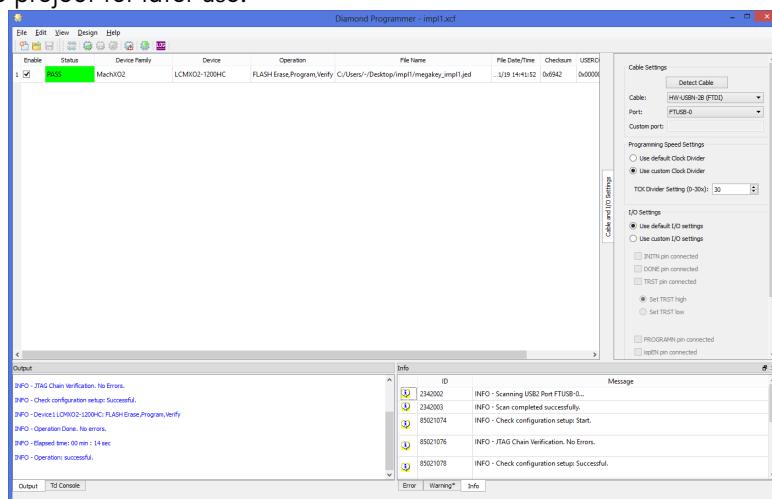
Step 9: Select cable:

After a moment the Output window should display "INFO - Operation: successful." and the "Status" cell should go green (does not always happen).



Step 10: Operation successful:

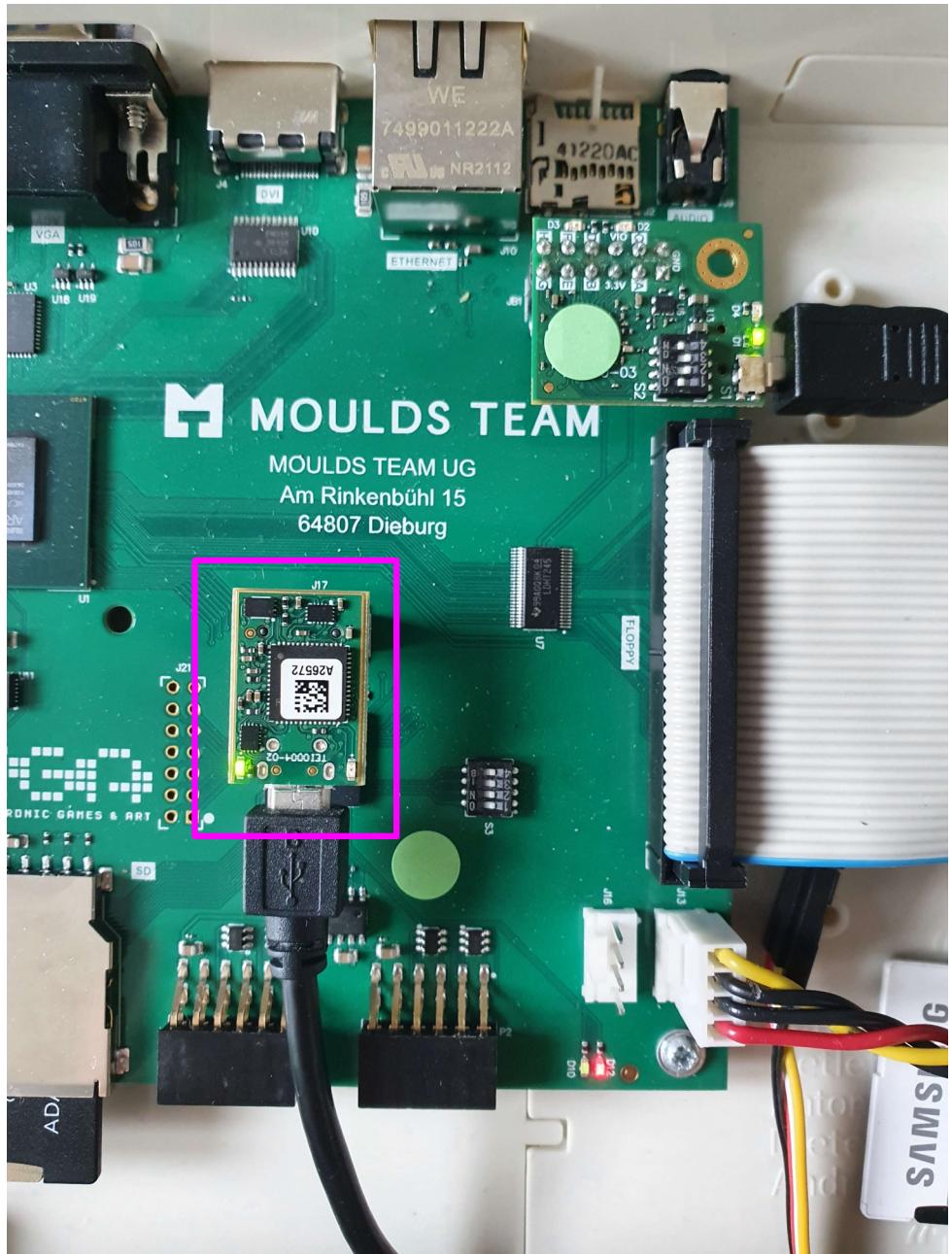
You have now successfully flashed the MEGA65 keyboard. If you wish you can now save the project for later use.



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.

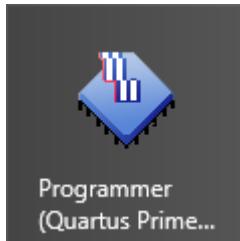
On the PCB R2/R3/R3A/R4 MEGA65 mainboard, all dip switches must be in the OFF position. The main FPGA of the MEGA65 must not contain a valid bitstream. The easiest way to do this is if you have a TE0790 JTAG adaptor: you can then use the m65 tool from a connected computer to begin sending a bitstream via JTAG, and then using control-C to abort the m65 tool before it can finish loading the bitstream.



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.

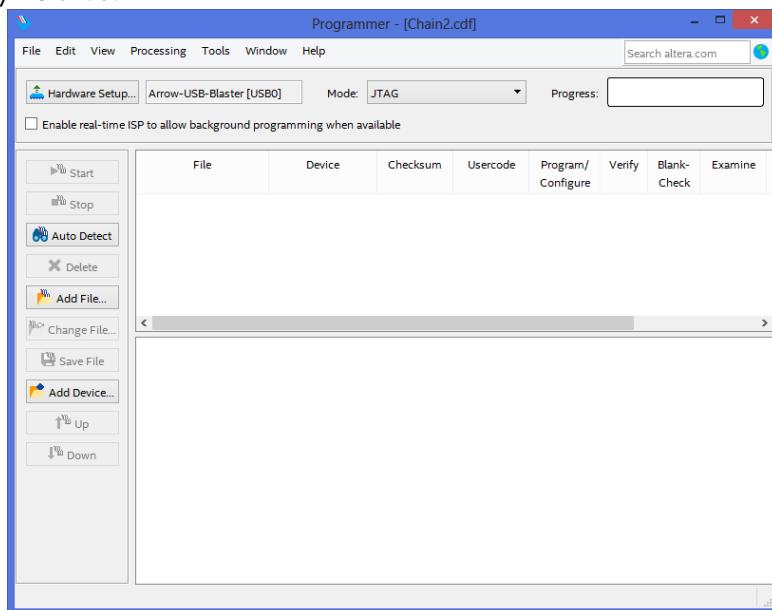
Step 1: Open Quartus Prime Programmer Lite Edition:

Click the "Hardware Setup" button in the top left corner of the Quartus Prime Programmer window.



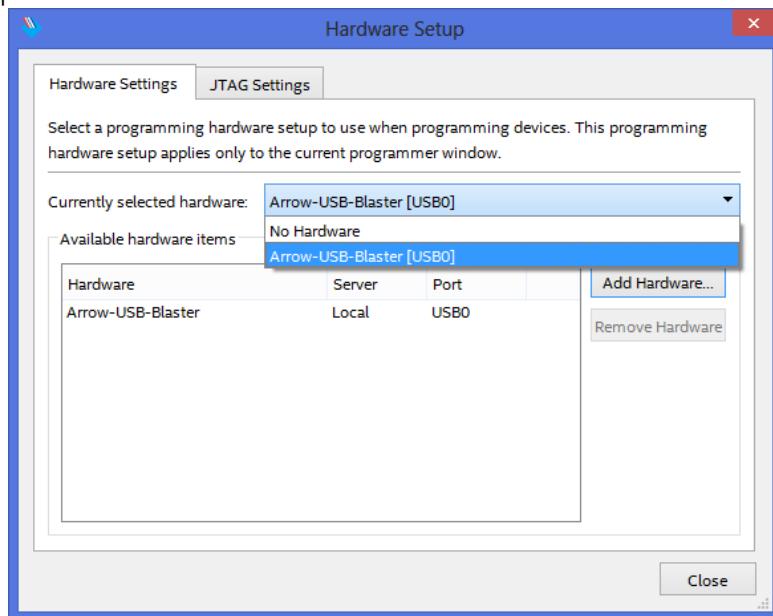
Step 2: Enter Hardware Setup:

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.



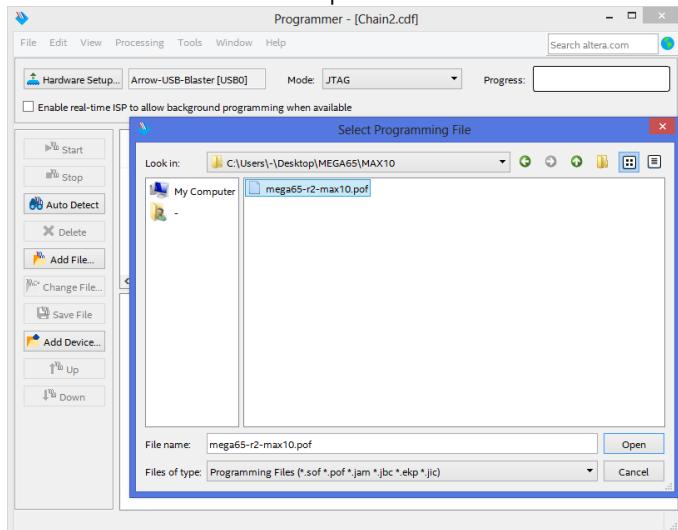
Step 3: Select Arrow USB-Blaster:

Click the "Add File" button from the left row and choose the latest ".pof" file. Then click "Open".

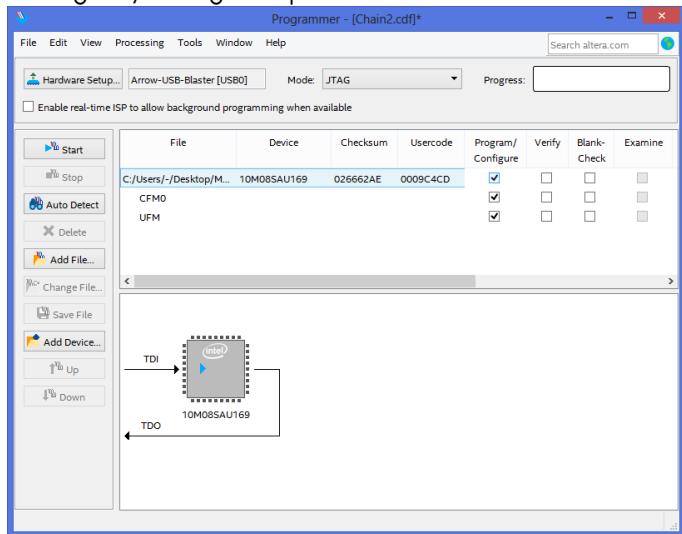


Step 4: Select Programming File:

Tick at least the three boxes under "Program/Configure". Also enabling all boxes under "Verify" and "Blank-Check" will make the process more reliable.



Step 5: Select Program/Configure Options:

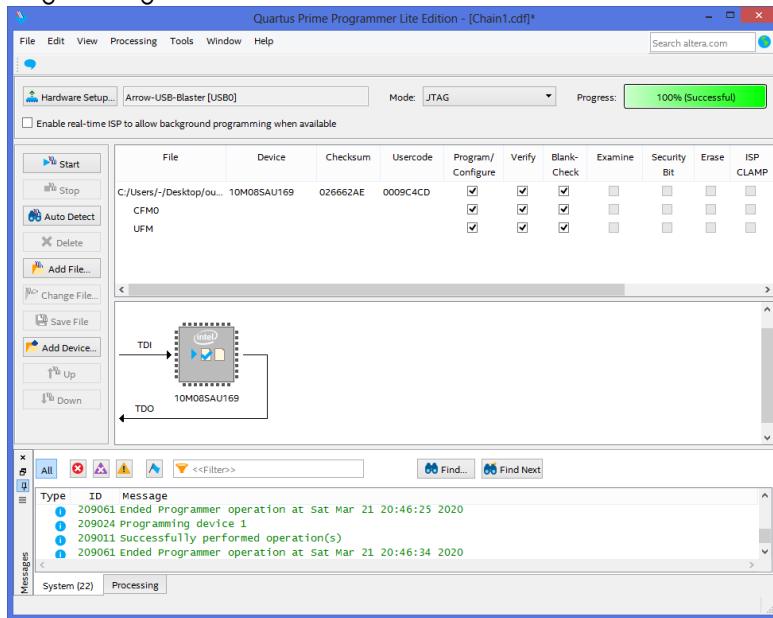


While keeping the Reset-Button pressed, switch the MEGA65 computer ON. The keyboard will go into "ambulance mode" (blue flashing lights). If it does not, the main FPGA is not empty – restart the whole process.

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 main FPGA bitstream has been erased and that you did not release the reset-button on the MEGA65 beforehand. Switch off the MEGA65 and restart this step.

Step 6: Programming successful:



U

APPENDIX

Trouble shooting

- **Hardware**
- **Vivado**
- **mega65_ftp**

HARDWARE

No lights when powering on

If there are occasions when your MEGA65 display any lights when powering on, they relate to having certain Digital Video devices plugged in while the MEGA65 is off, that don't provide enough power for the keyboard's CPLD to be properly powered on, but enough to stop it properly resetting when the MEGA65 powers on. Removing the Digital Video cable and switching the machine off and on again fixes the issue.

VIVADO

RAM requirements

```
INFO: [Synth 8-256] done synthesizing module 'ram32x1024' [/home/....]
INFO: [Synth 8-256] synthesizing module 'charrom' [/home/....]
/opt/Xilinx/Vivado/2019.2/bin/loader: line 280: 2317 killed
WARNING: [Vivado 12-8222] Failed run(s) : 'synth\_1'
ERROR: Application Exception: failed to launch run 'impl\_1'
      due to failures in the following run(s):
      synth\_1
These failed run(s) need to be reset prior to launching 'impl\
      _1' again.
```

This error is due to Vivado crashing because the machine doesn't have enough RAM for Vivado to run. Vivado requires at least 4GB to synthesise the MEGA65 target, but 8GB is better.

MEGA65_FTP

Missing Library

```
/usr/bin/ld: cannot find -lncurses  
collect2: error: ld returned 1 exit status  
Makefile:474: recipe for target 'bin/mega65_ftp' failed  
make: *** [bin/mega65_ftp] Error 1
```

This error occurs when the ncurses library is missing from the computer when building the mega65_ftp program. To rectify this issue you will need to ensure that you install this dependency.

```
sudo apt-get install libncurses5-dev libncursesw5-dev
```

V

APPENDIX

Model Specific Features

- Detecting MEGA65 Models
- MEGA65 Desktop Computer, Revision 3
onwards
- MEGA65 Desktop Computer, Revision 2
- MEGAphone Handheld, Revisions 1 and 2
- Nexys4 DDR FPGA Board

DETECTING MEGA65 MODELS

While we expect the production version of the MEGA65 to be a stable platform, there may still be cases where detecting which hardware your program is running on. This is particularly important for the MEGA65 system software, which may need to initialise different pieces of hardware on the different models. Also, because there is a hand-held version of the MEGA65 already in development, which uses a slightly different resolution screen (800x480 instead of 720x576), and has a touch screen but no hardware keyboard, you may wish to make programs that adapt to the hand-held devices in a more graceful way. For example, you may enable touch-screen input, and restructure on-screen selections to be large enough to be easily activated by a finger.

The simple way to detect which model of MEGA65 your program is running on, is to check the \$D629 register (but don't forget to enable the MEGA65 I/O personality first, via \$D02F). This contains an 8-bit hardware identifier. The following values are currently defined:

\$01 (1) MEGA65 R1

\$02 (2) MEGA65 R2

\$03 (3) MEGA65 R3

\$21 (33) MEGAphone (hand-held) R1

\$40 (64) Nexys4 PSRAM

\$41 (65) Nexys4DDR

\$42 (66) Nexys4DDR with widget board

\$FD (253) QMTECH Wukong A100T board

\$FE (254) Simulation run of VHDL

MEGA65 DESKTOP COMPUTER, REVISION 3 ONWARDS

The R3 desktop PCB is very similar to the R2 desktop PCB, with two key changes:

- First, the R3 PCB does not have an ADV7511 digital video driver chip, and so the I2C register block for that device is not present.
- Second, the R3 PCB uses a different on-board amplifier for the PC speakers, which are now present in stereo, rather than mono as on the R2 PCB. The ampli-

fier on the R3 PCB is the same as on the MEGAphone R1 - R2 PCBs. However, the I2C registers are at a different address. On the MEGA65 R3 PCB, the registers are located at \$FFD71DC - \$FFD71EF.

MEGA65 DESKTOP COMPUTER, REVISION 2

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 65 in C65-mode):

```
10 RESTORE 110:FOR I=0TO43:READ A$:POKE 1024+I,DEC(A$):NEXT: BANK 128:SYS 1042
20 S=PEEK(1056):M=PEEK(1057):H=PEEK(1058)
30 D=PEEK(1059):MM=PEEK(1060):Y=PEEK(1061)+DEC("2000")
40 IF H AND 128 GOTO 80
50 PRINT "THE TIME IS ";RIGHT$(HEX$(H AND 63),2);":";
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,80,FF,81,00,00,00,08,00,10,71,0D,20,04,00,00,00,00
120 DATA A9,47,8D,2F,D0,A9,53,8D,2F,D0,A9,00,8D,02,D7,A9
130 DATA 04,8D,01,D7,A9,00,8D,05,D7,60
```

This program works by setting up a DMA list in memory at 1,024 (\$0400) (unused normally on the C65), followed by a routine at 1,042 (\$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 (\$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  
...  
...
```

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 I/O expanders, that are used to connect the joy-pad and other peripherals. The I/O expanders are fully integrated into the MEGAphone design, and thus there should be no normal need to read these registers directly. The I/O 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

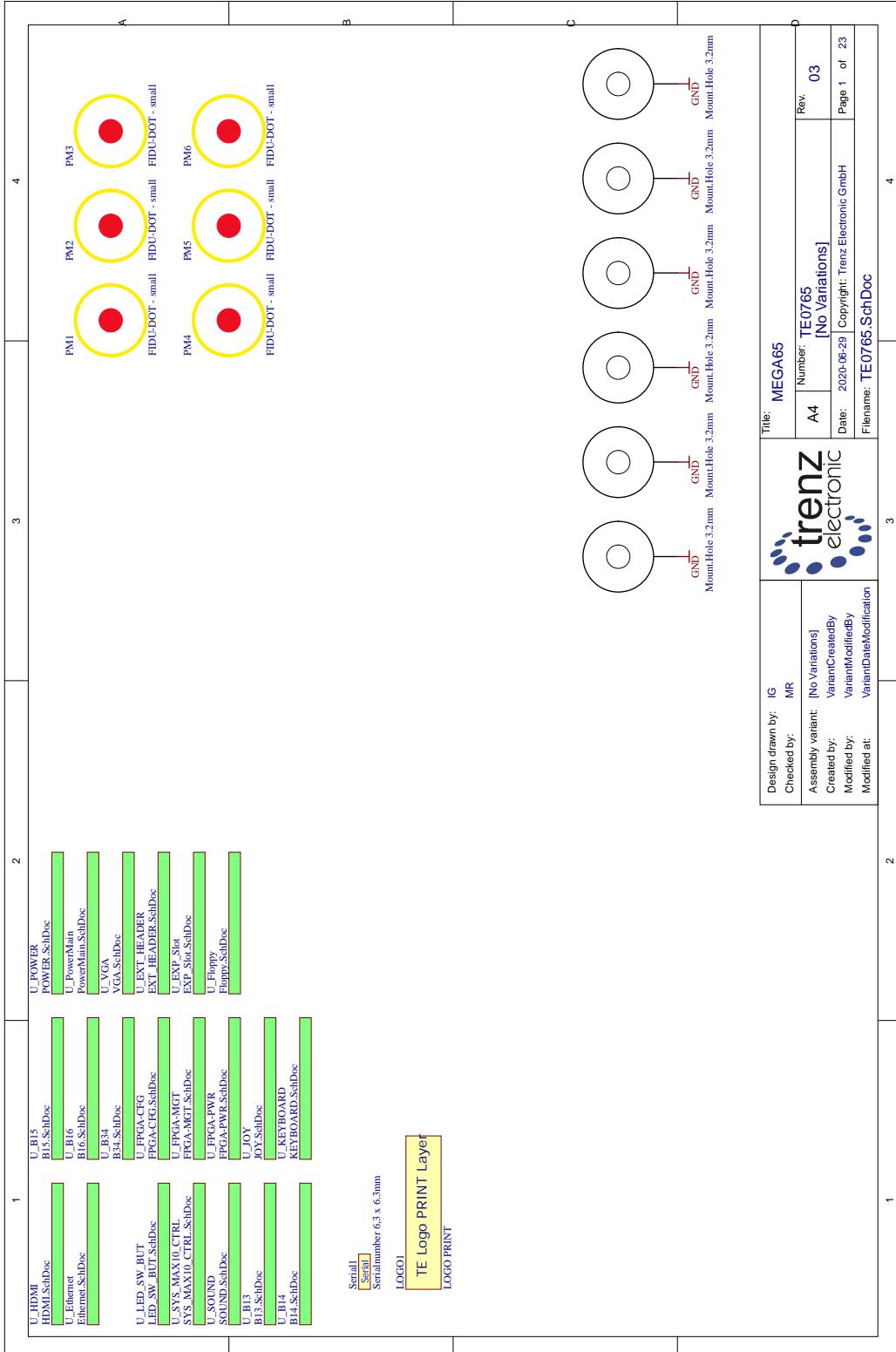
NOTE: These registers are not yet fully documented.

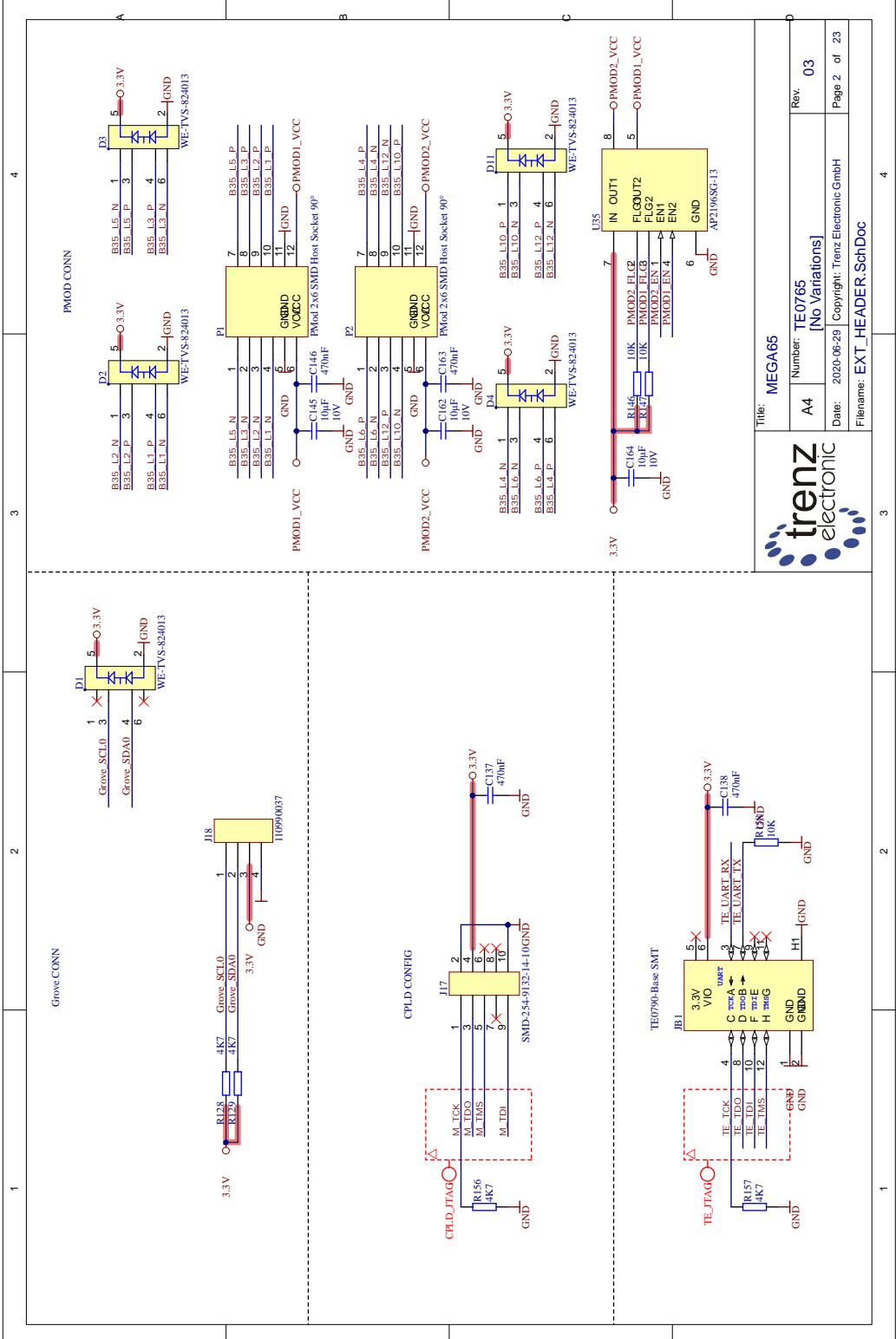
APPENDIX W

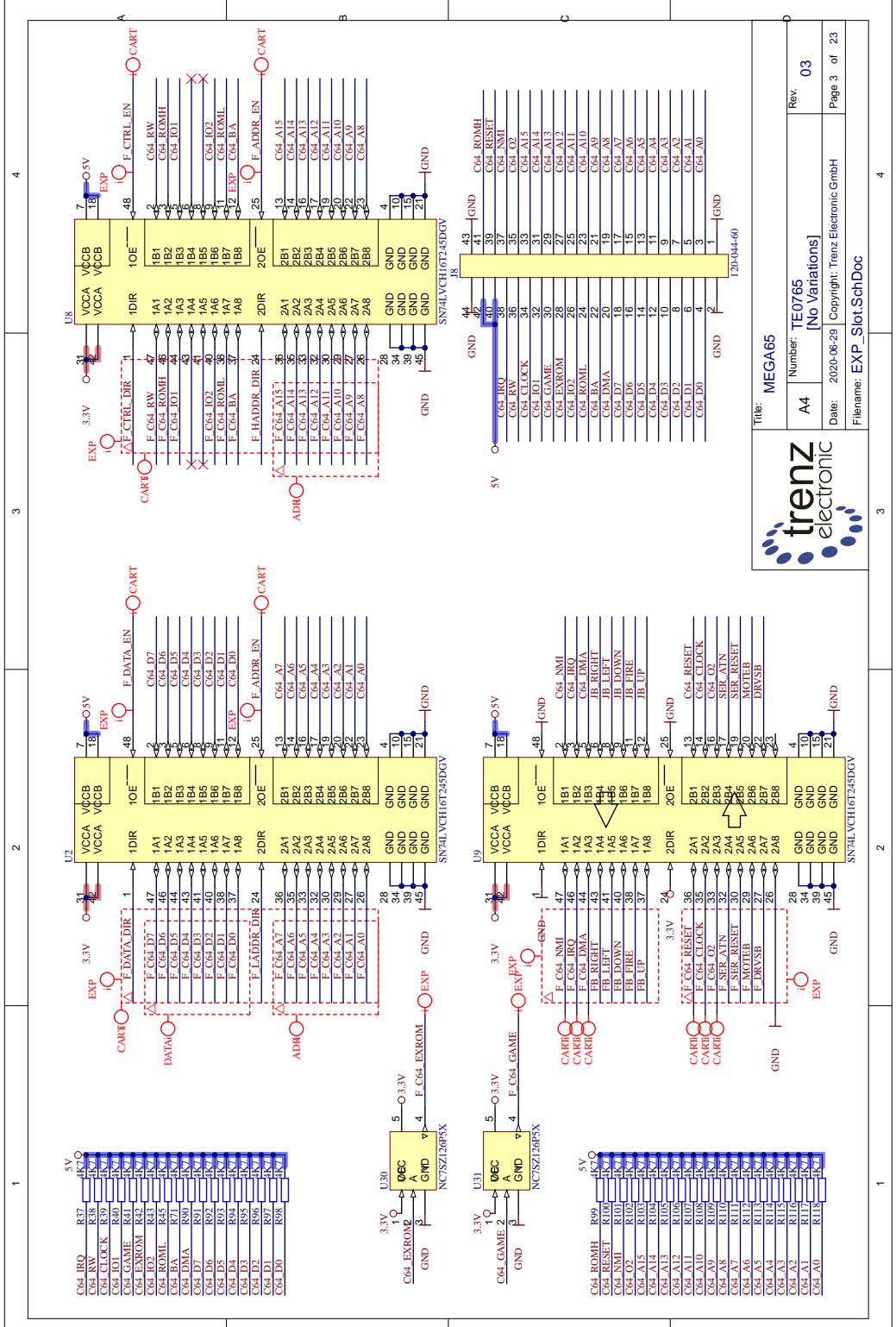
Schematics

- **MEGA65 R3 Schematics**
- **MEGA65 R2 Schematics**
- **Nexys Widget Board Schematics**

MEGA65 R3 SCHEMATICS







4

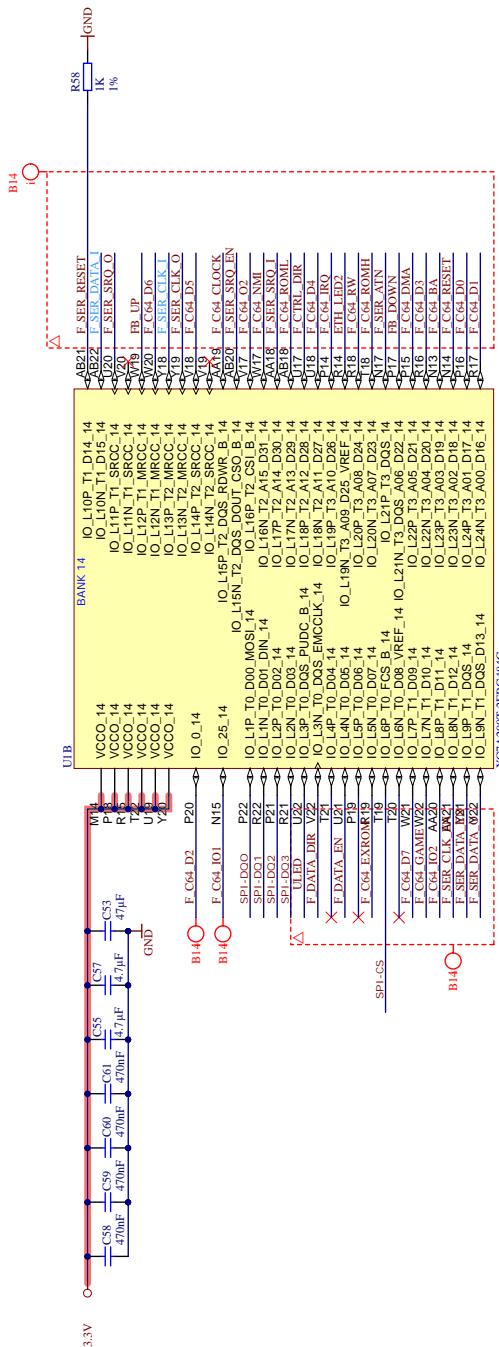
3

2

1

A

B14



C



C

Title: MEGA65
A4 Number: TE0765
[No Variations]
Date: 2020-06-29
Filename: B14.SchDoc

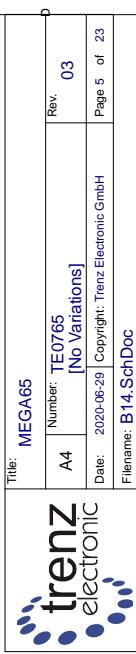
A4	Number: TE0765 [No Variations]	Rev. 03
Date: 2020-06-29	Copyright: Trenz Electronic GmbH	Page 5 of 23

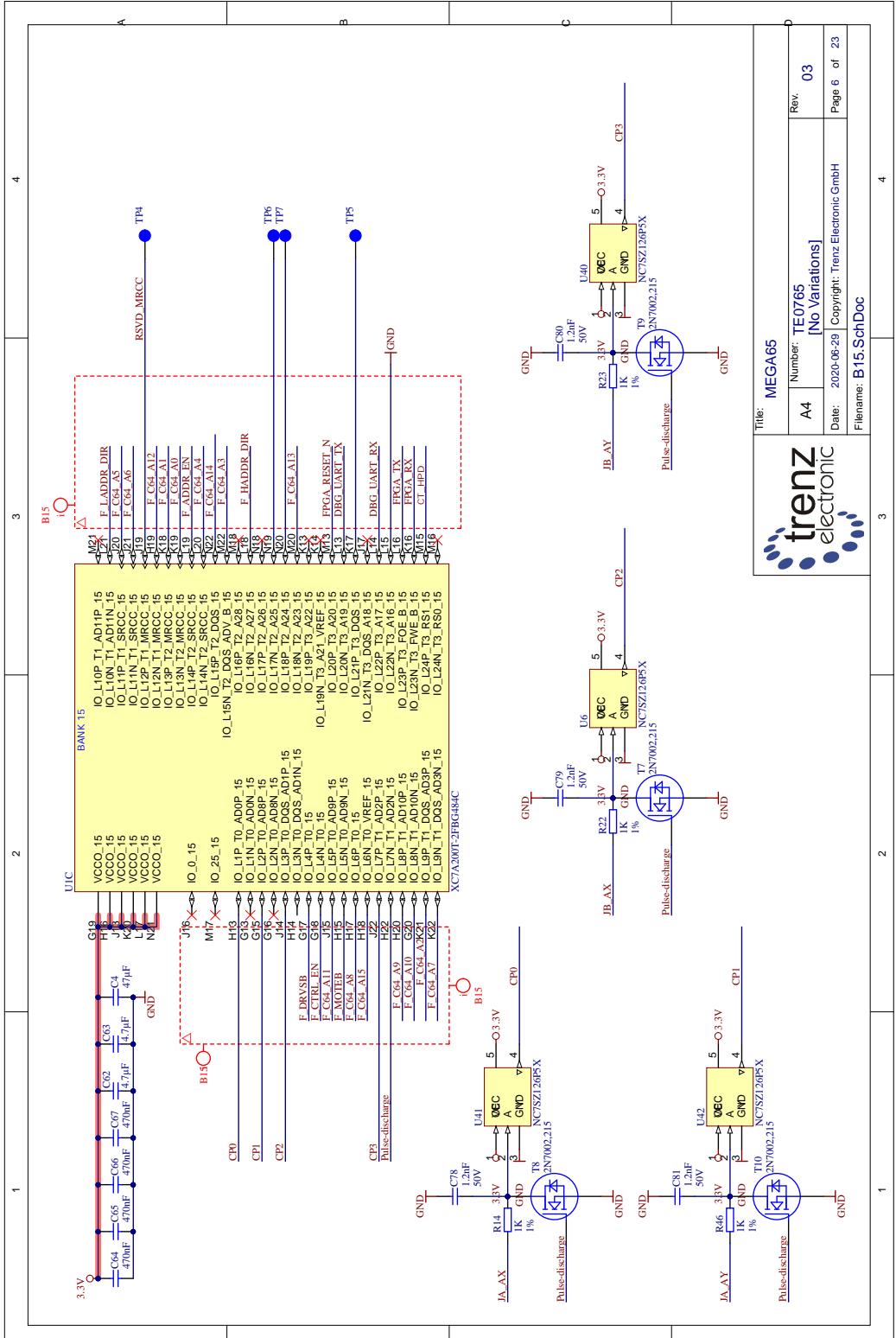
4

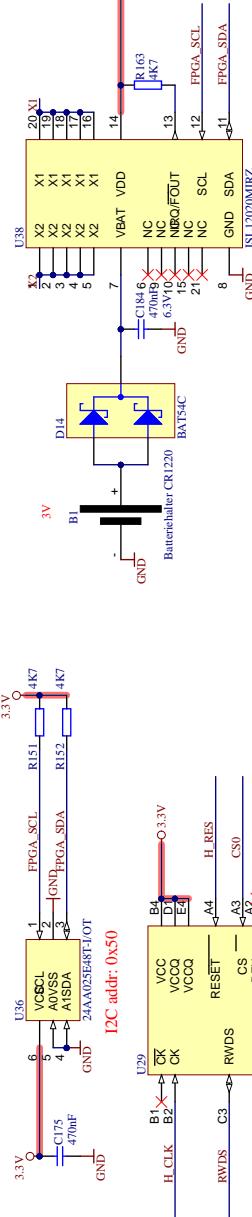
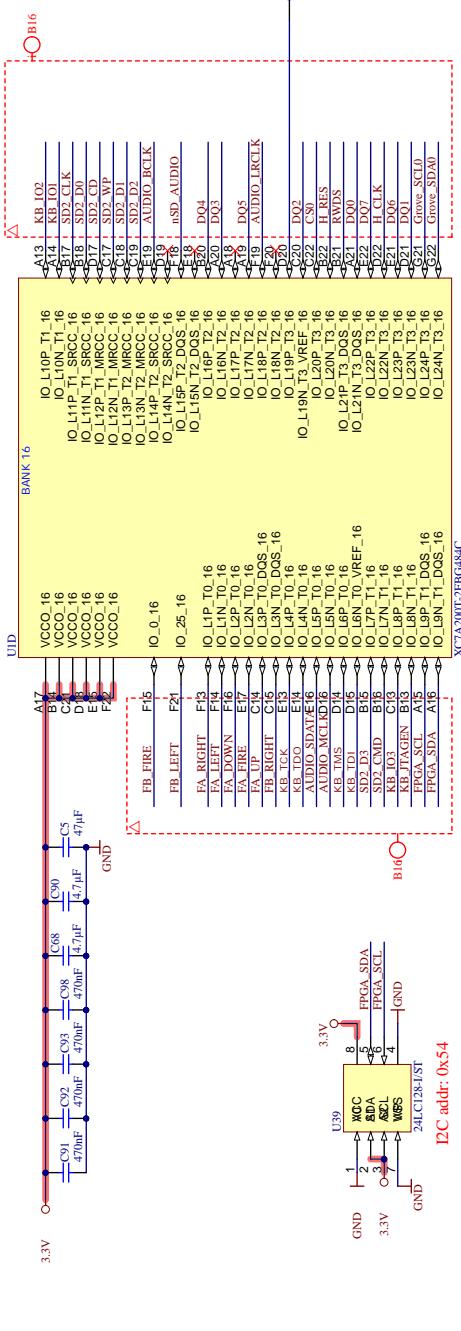
3

2

1





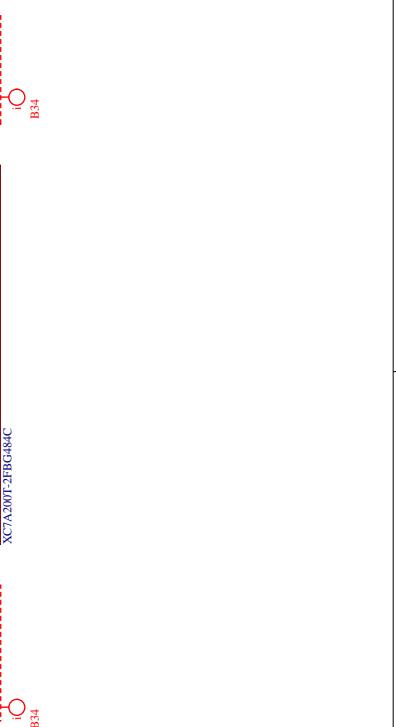
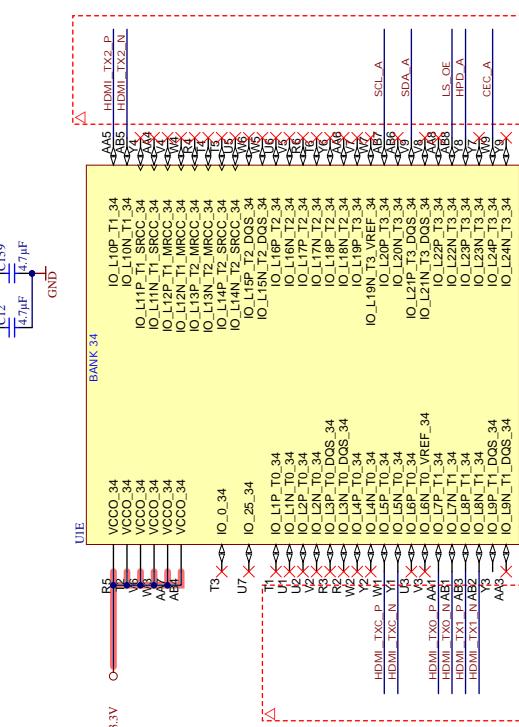
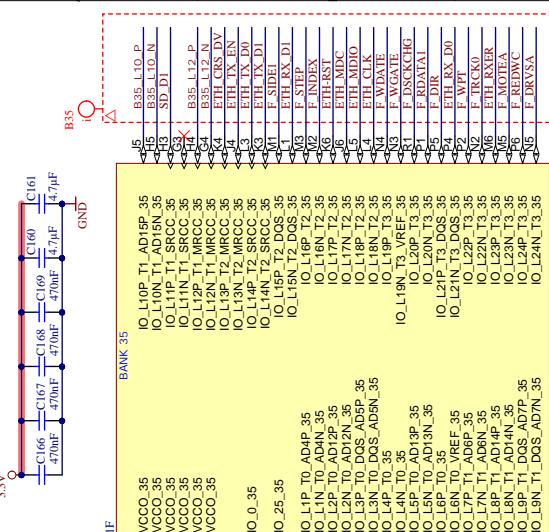


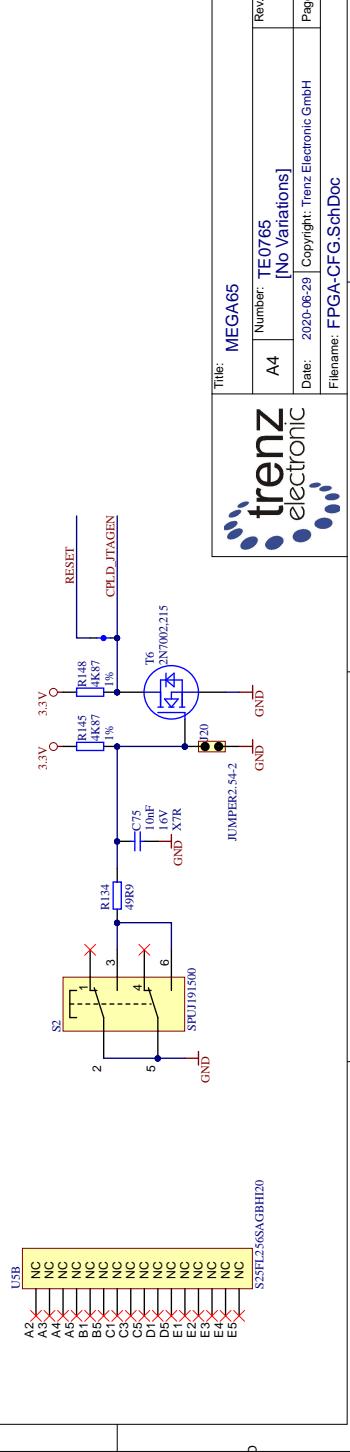
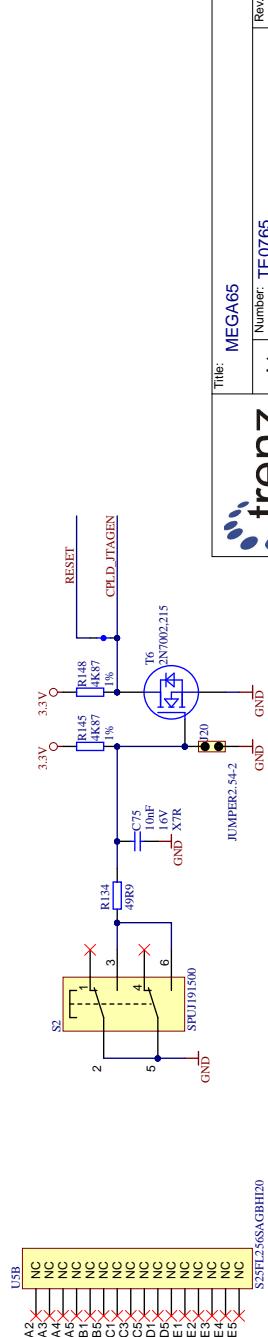
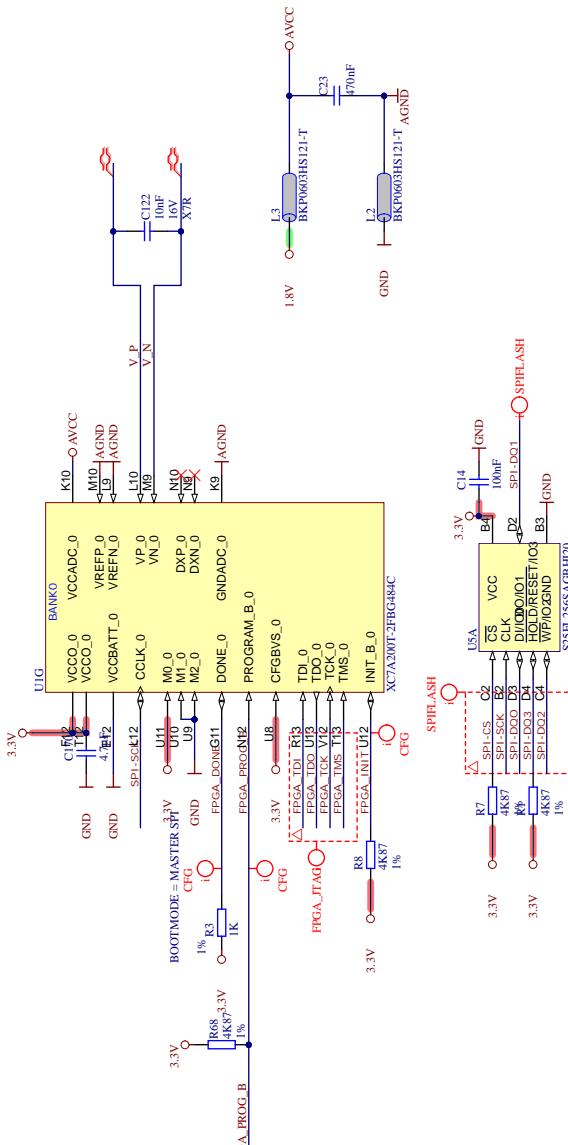
Title: MEGA65

A4 Number: TE0765
[No Variations]
date: 2020-06-29 Copyright: Tenz Electronic GmbH
filename: B16.SchDoc

Rev. 03
Page 7 of 23







4

3

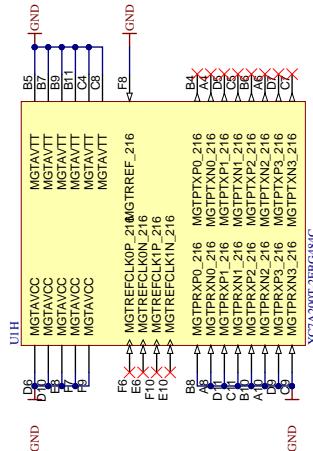
2

1

A

B

C



Title: MEGA65

A4	Number: TE0765 [No Variations]	Rev. 03
Date: 2020-06-29	Copyright: Trenz Electronic GmbH	Page 10 of 23
Filename: FPGA-MGT-SchDoc		

4

3

2

1

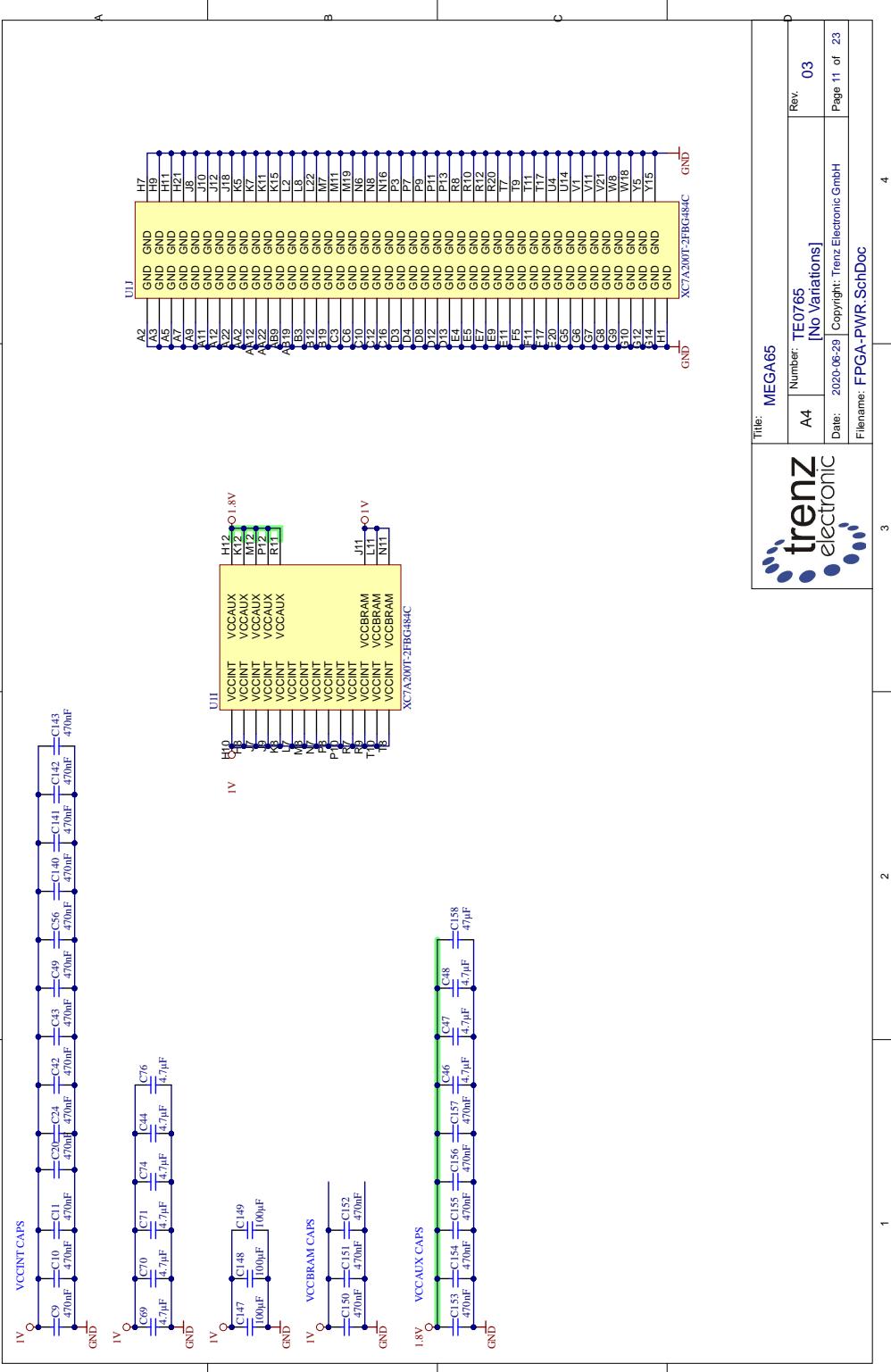
A

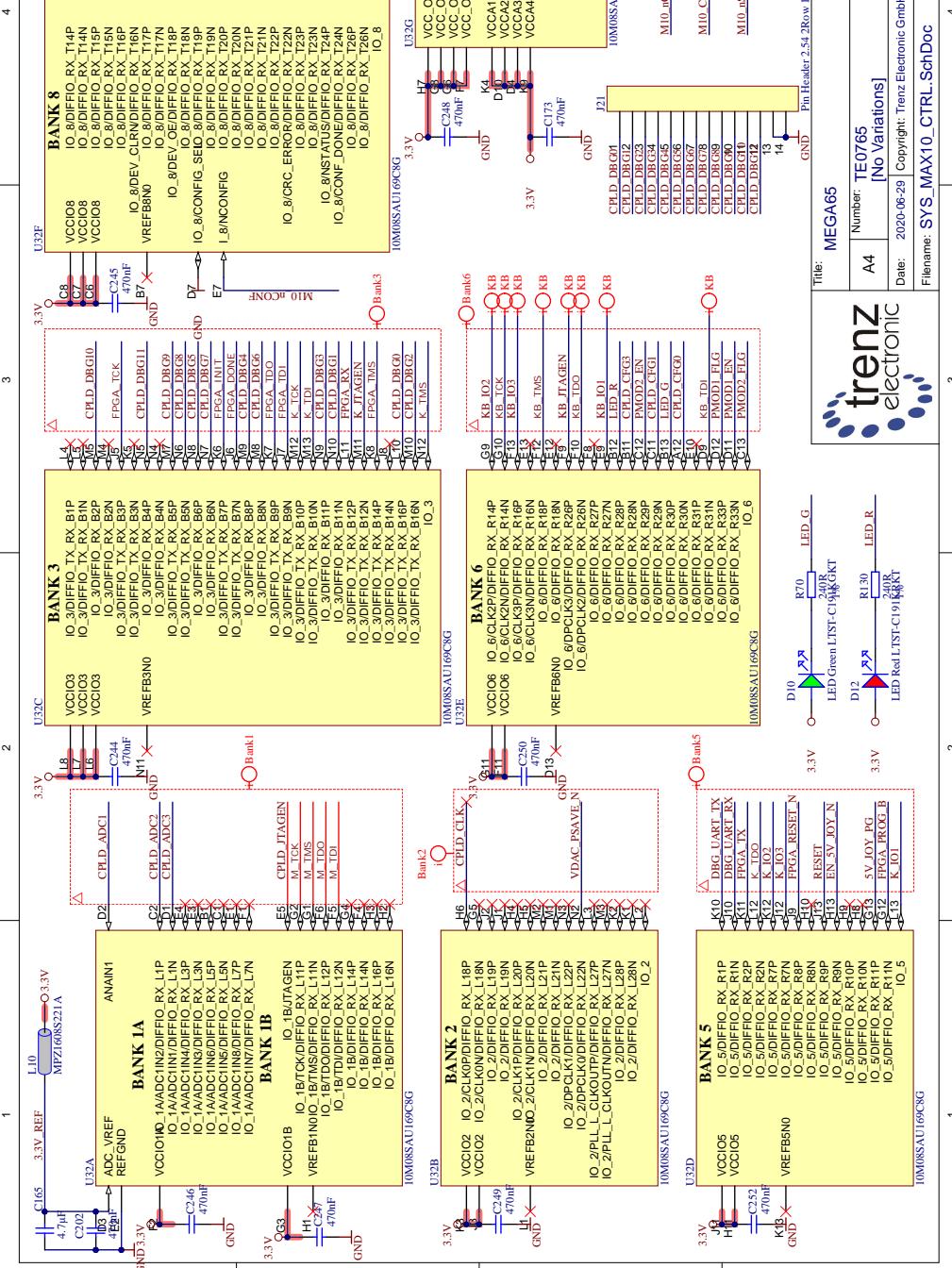
B

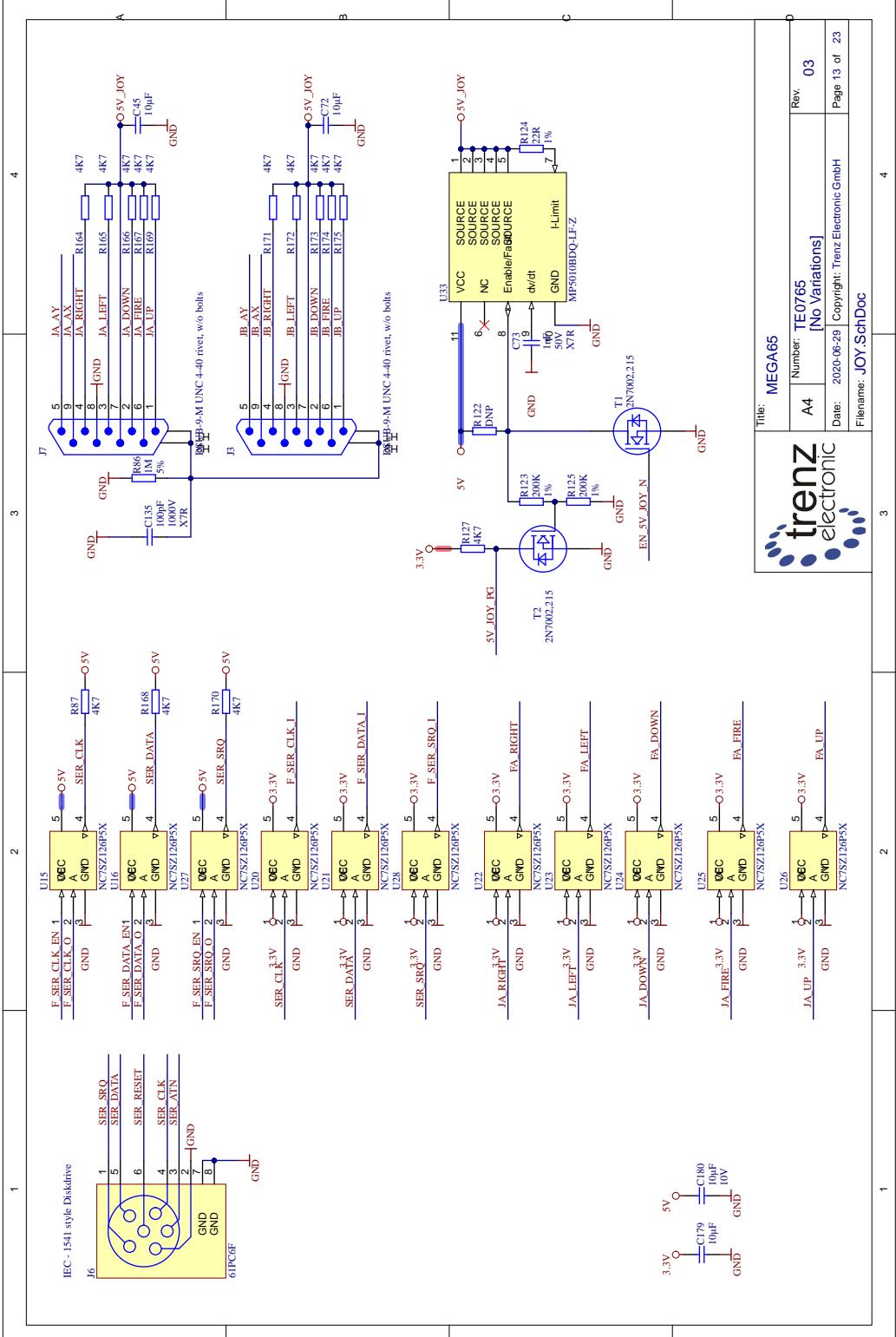
C

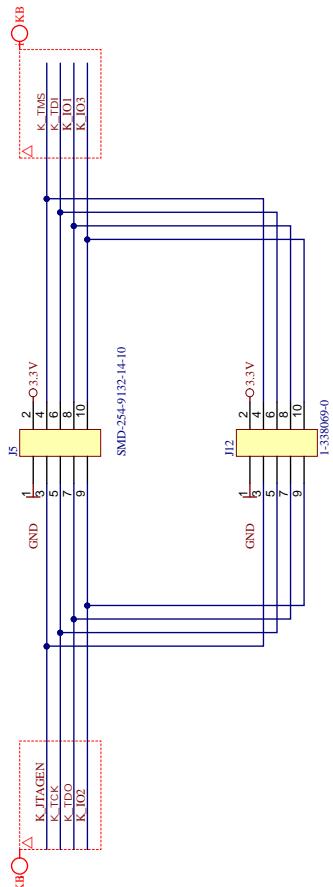
D

4









1

2

3

4

A

A

B

B

C

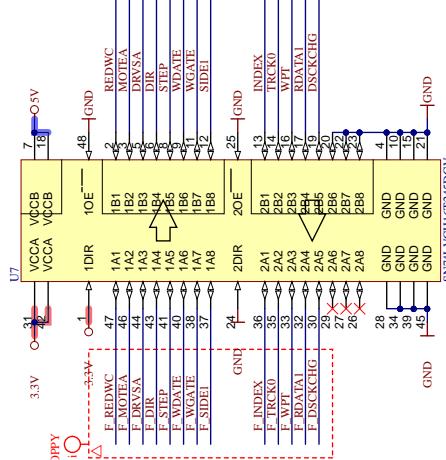
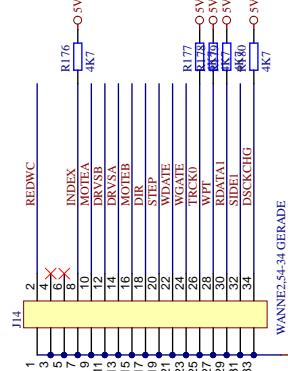
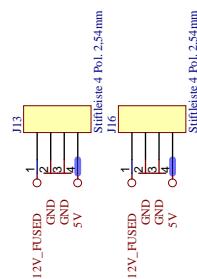
C

1

2

3

4



Title: MEGA65
A4 Number: TE0765
[No Variations]
Rev. 03
Date: 2020-06-29 Copyright: Trenz Electronic GmbH
Page 15 of 23
Filename: Floppy.SchDoc



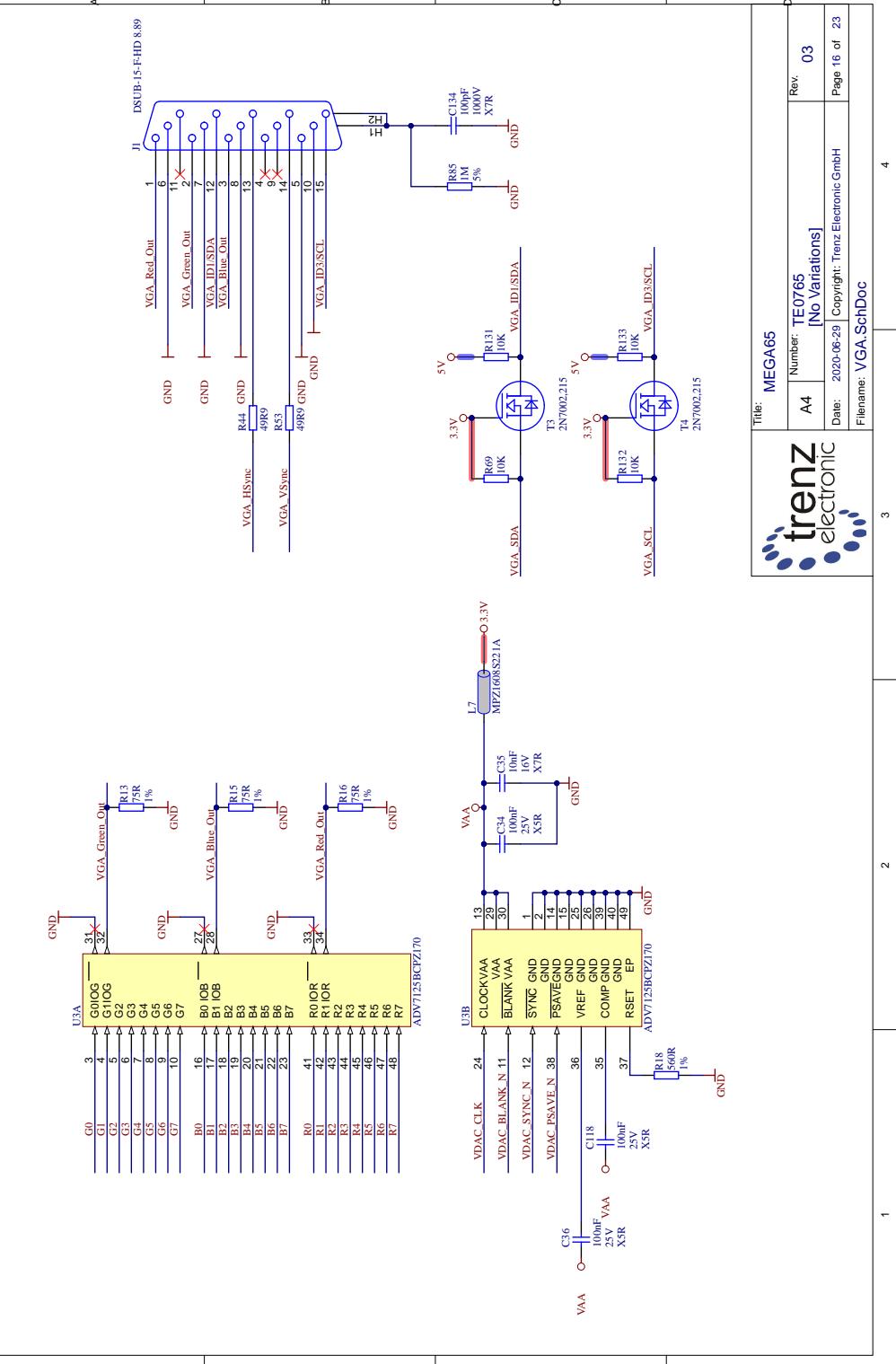
D

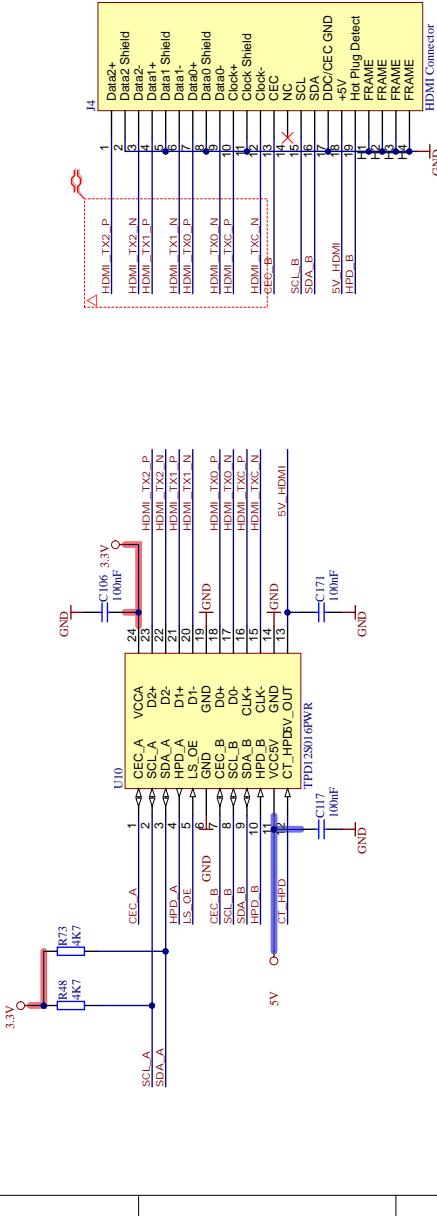
E

2

3

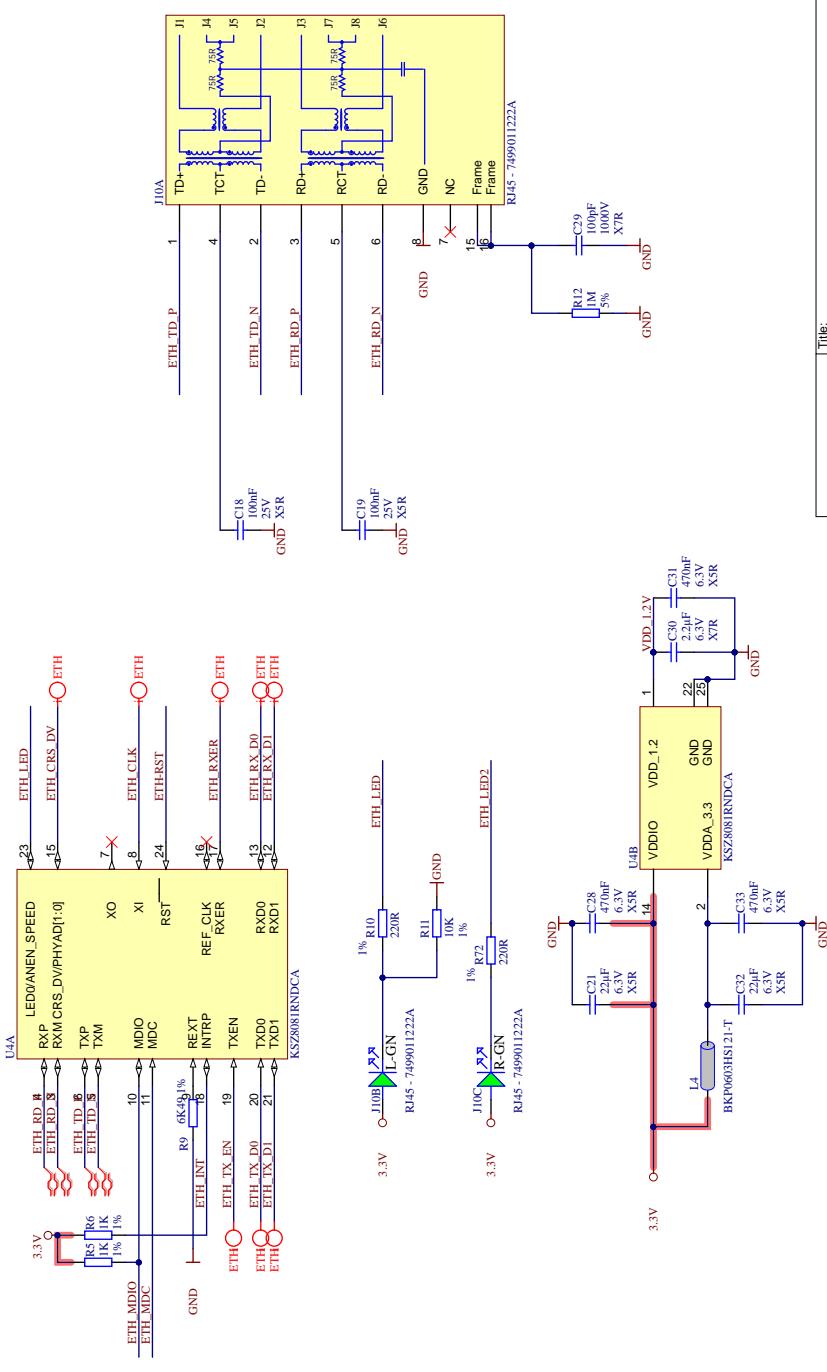
4





Title: MEGA65

A4	Number: TE0765 [No Variations]	Rev. 03
Date: 2020-06-29	Copyright: Trenz Electronic GmbH	Page 17 of 23
Filename: HDMI_SchDoc		



lei: MEGA65

Number:

第4章

Username: Etherr
Date: 2020-06-2

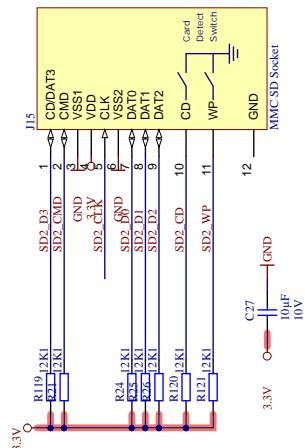
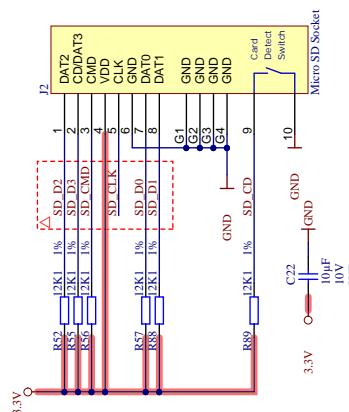
4

3

2

1

A

http://linux-sunxi.org/MicroSD_Breakout

A

B

C

Title: MEGA65

A4 Number: TE0765 [No Variations]

Rev. 03

Date: 2020-06-29 Copyright: Trenz Electronic GmbH

Page 19 of 23

Filename: LED_SW_BUT.SchDoc

4

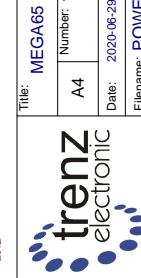
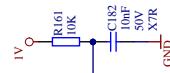
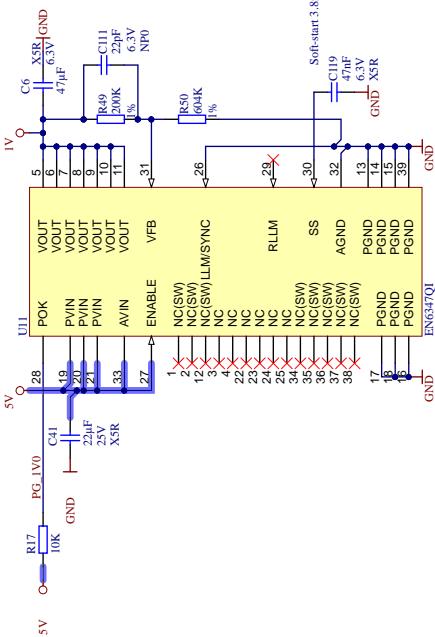
3

2

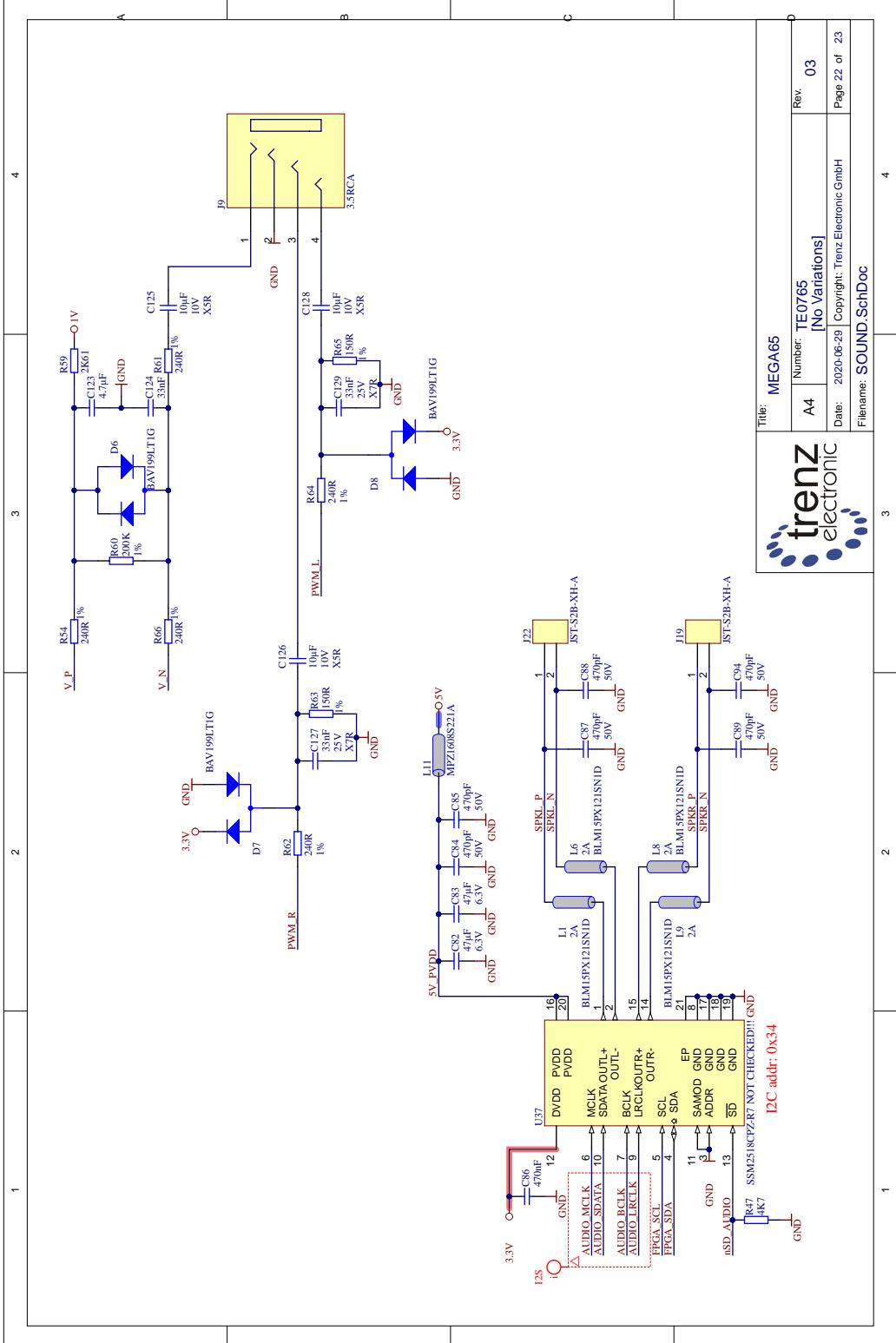
1

D





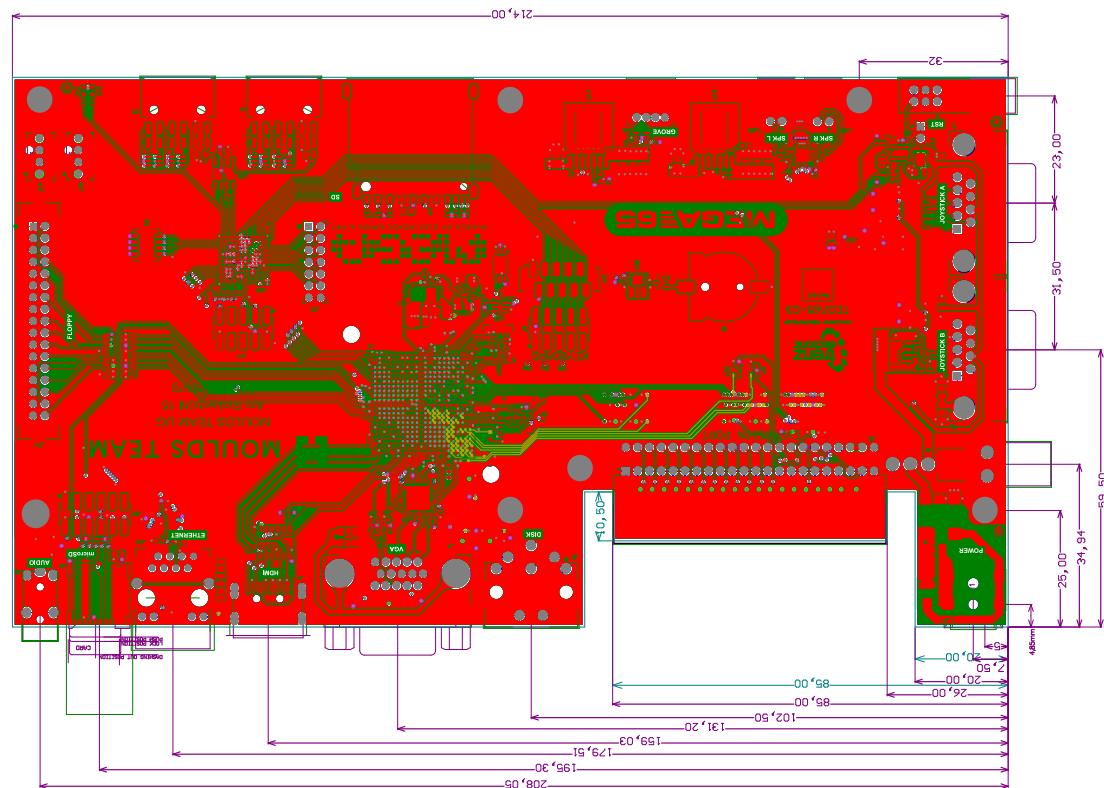
trenz electronic
Title: MEGA65
A4 Number: TE0765
[No Variations]
Rev. 03
Date: 2020-06-29 Copyright: Trenz Electronic GmbH / TT
Page 21 of 23
Filename: POWER.SchDDoc



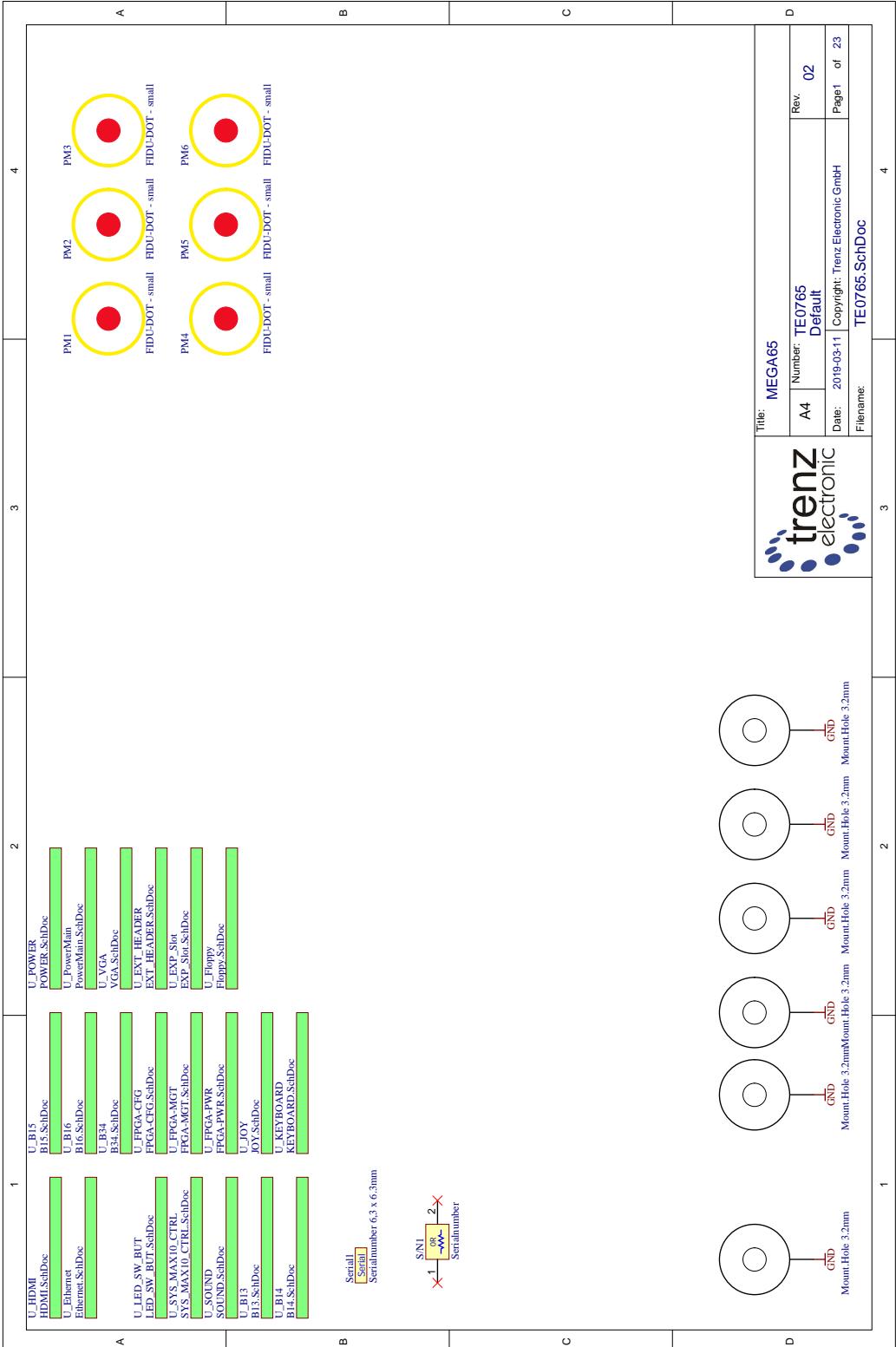
A	1	2	3	4
B				
C				
D				
REVISION HISTORY				
REV	Description	1G		
-03	<p>1. Added a VRP resistor on bank 63;</p> <p>2. LDO U133 is changed on ADP7102ACPZ;</p> <p>3. Signal FPGA Q00 is connected on AE18 pin of FPGAs;</p> <p>4. Signal DIG, LED3 is connected on AD18 pin of FPGAs;</p> <p>5. Signal MIO13_25 connected to J1 pin 33 instead MIO25;</p> <p>6. Resistor R84 is removed;</p> <p>7. LED D1 moved on edge of PCB;</p> <p>8. Added TH1 resistors J4 on CPLD_ITAGEN, R76 was removed;</p> <p>9. Signals B9_Y, X are renamed in B88_XX_X;</p> <p>10. C241 is changed on inf;</p> <p>11. Length of CLK signals on RFADC and RFDAC are adjusted;</p> <p>12. Wrong connection R83 on back of PCB;</p> <p>13. Wrong connection PGOOD1 pin of U7 is fixed;</p>			
A	1	2	3	4

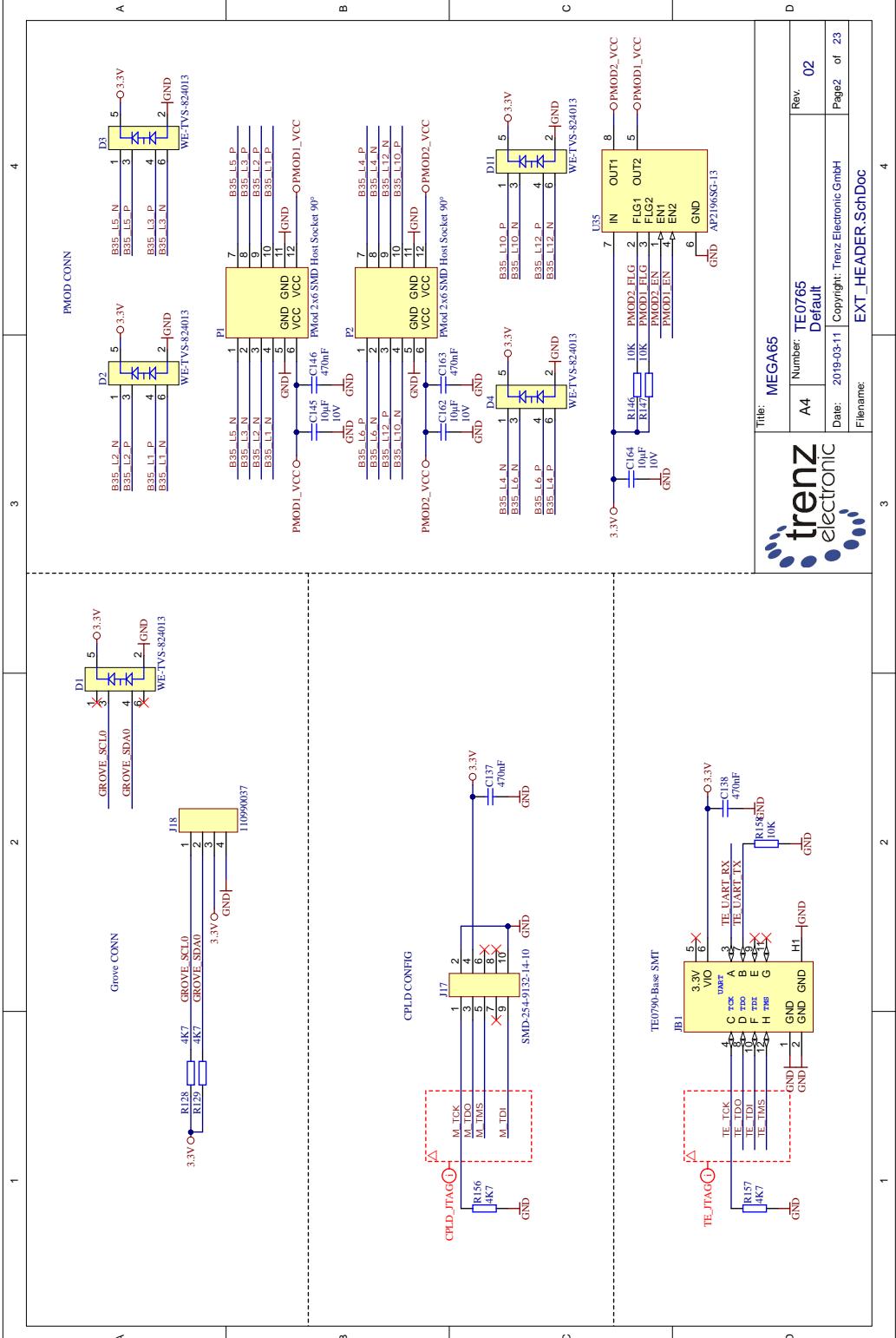


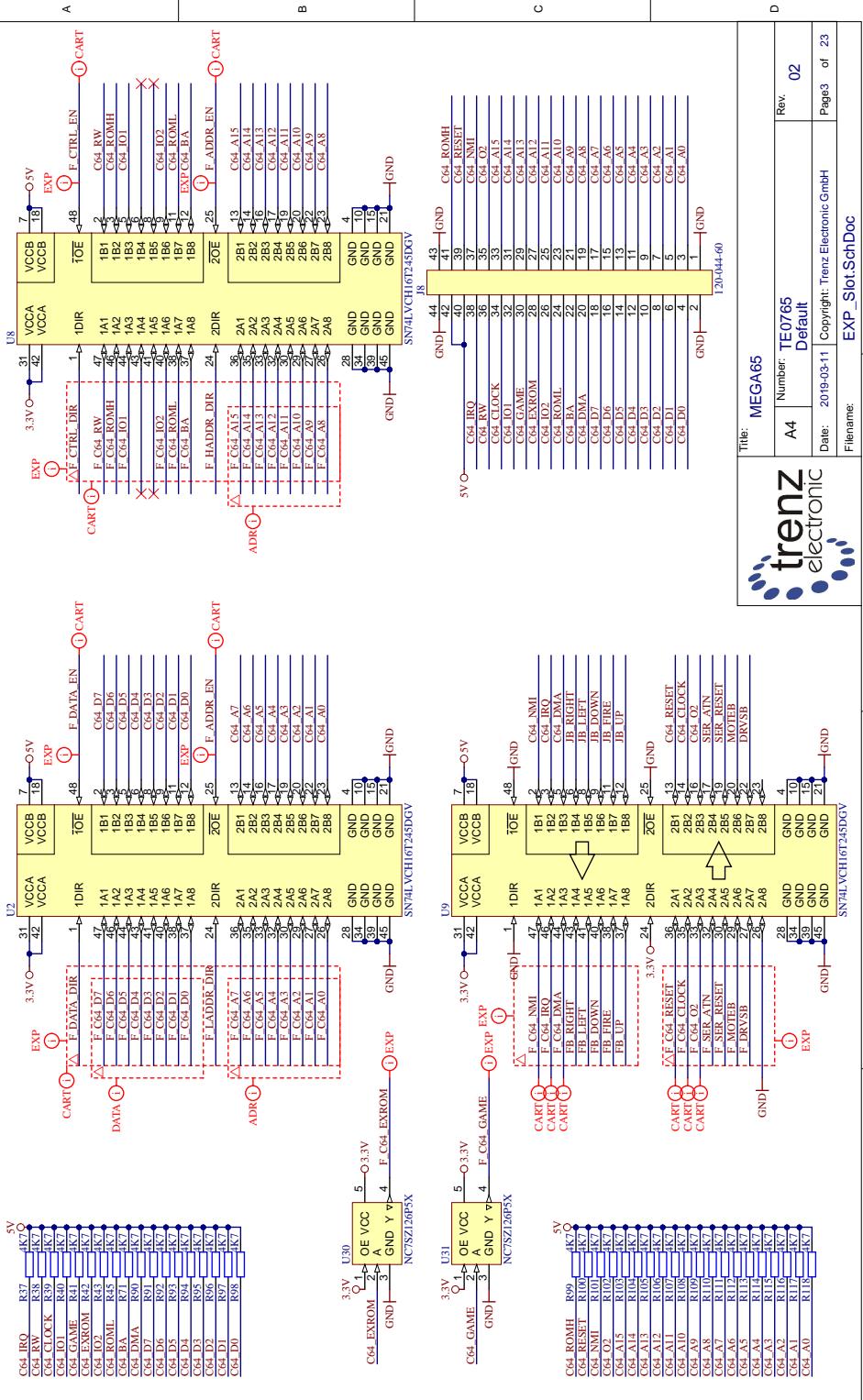
Title: MEGA65
 A4 Number: TE0835 [No Variations]
 Date: 2020-06-29 Rev. 03
 Copyright: trenz Electronic GmbH
 Page 23 of 23
 Filename: Revision_Changes_SchDoc

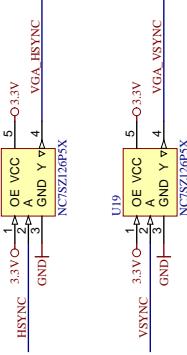
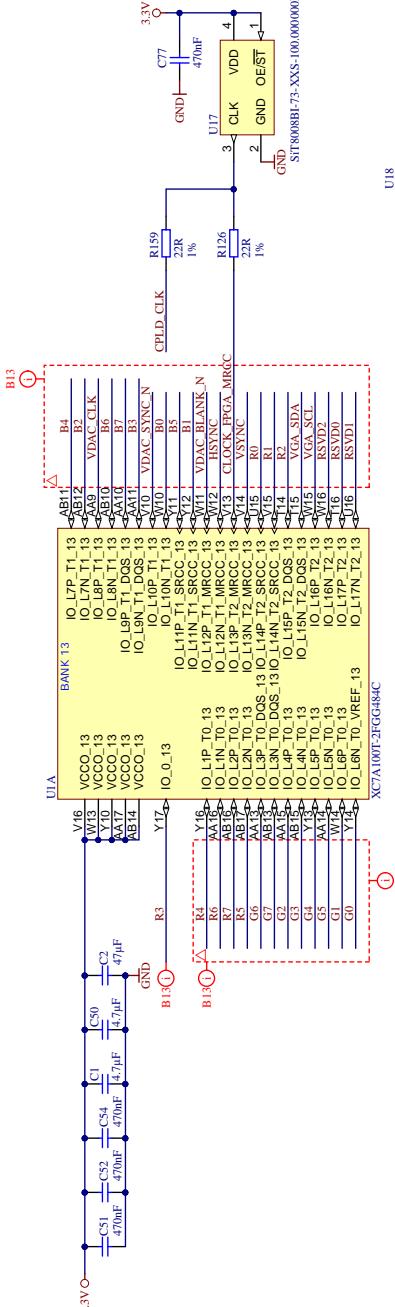


MEGA65 R2 SCHEMATICS









MEGA65

D
Rev.

02

Page 4 of 23

ANSWER

A

A

B

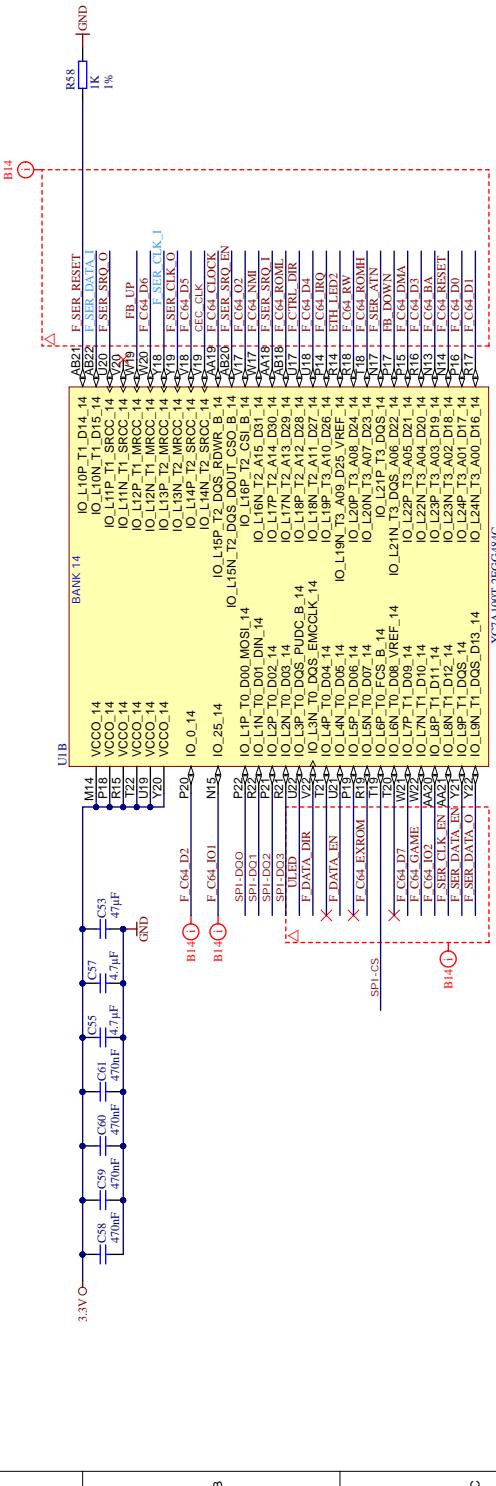
B

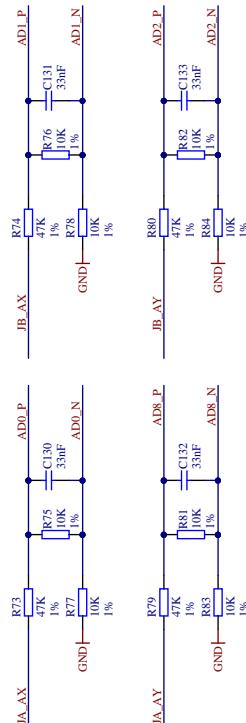
C

C

D

D

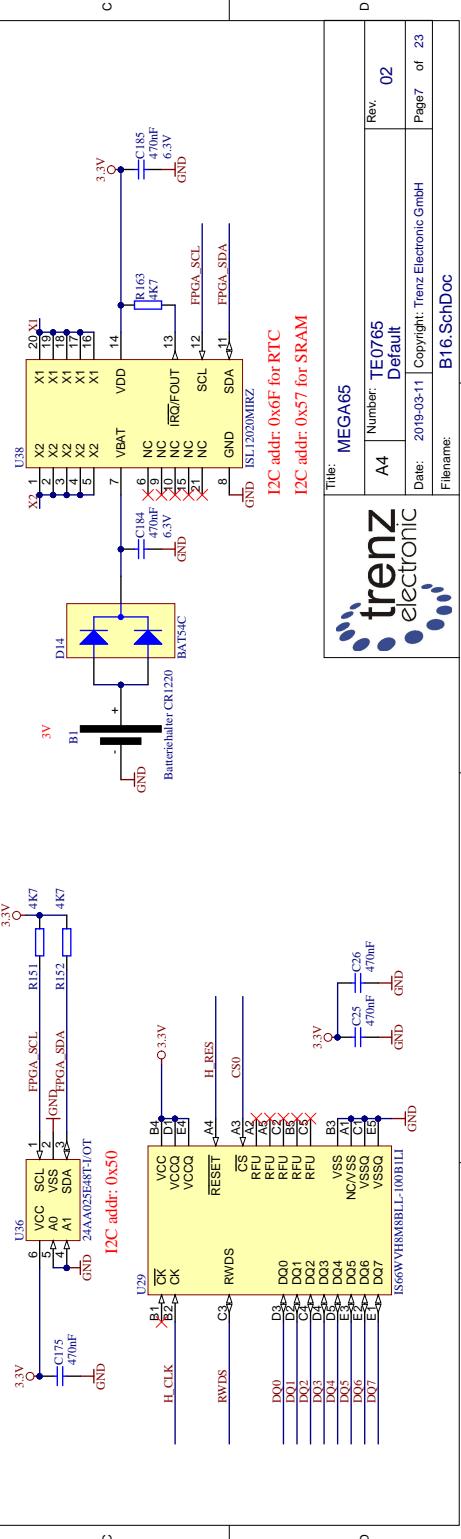
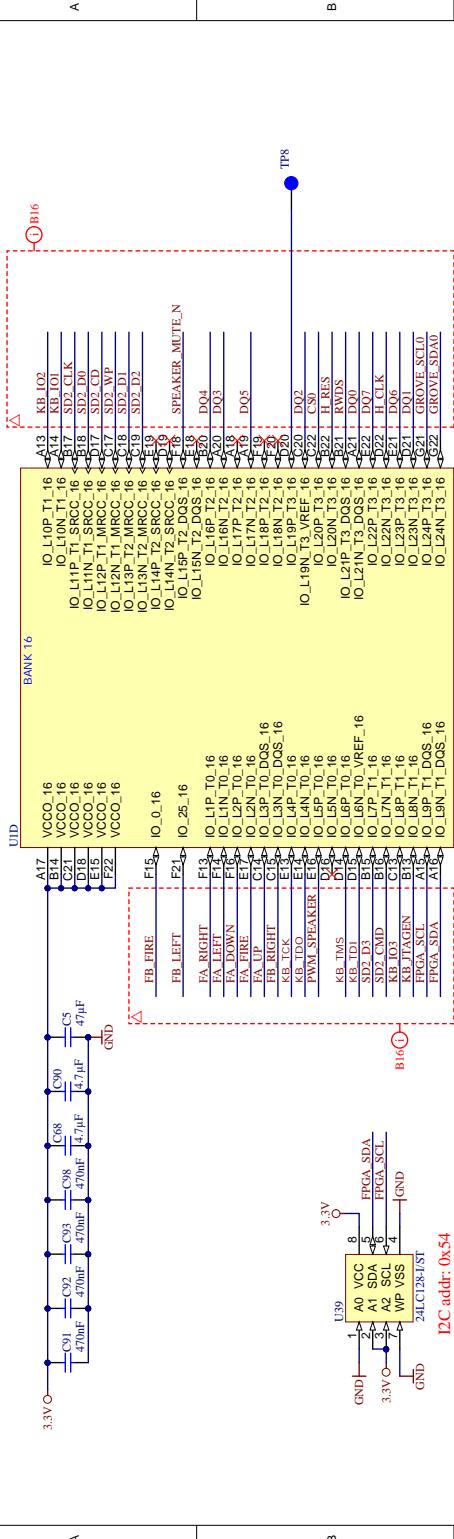




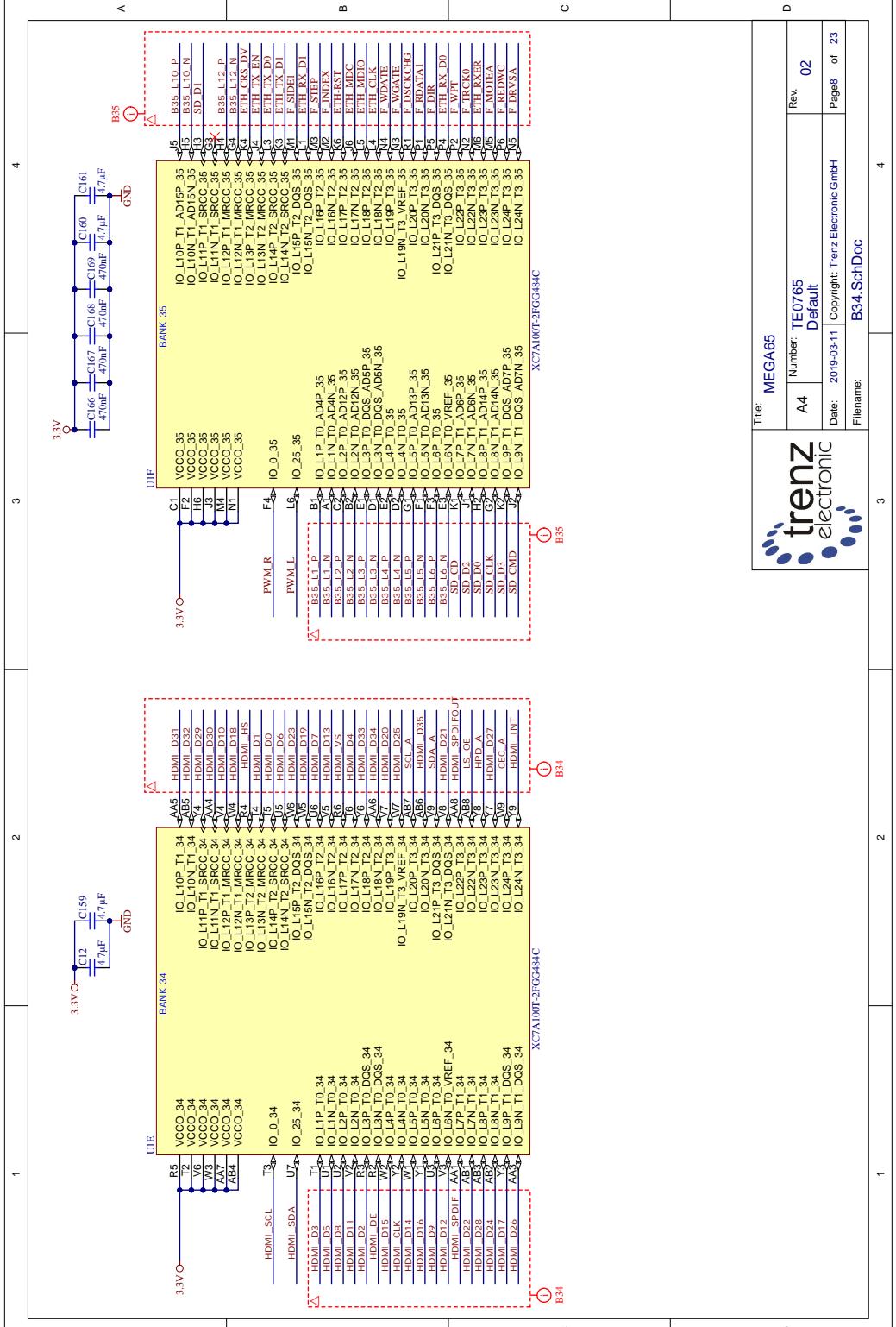
Title:	MEGA65
A4	Number:
Date:	2019-03-31
Filename:	trenz_electronic

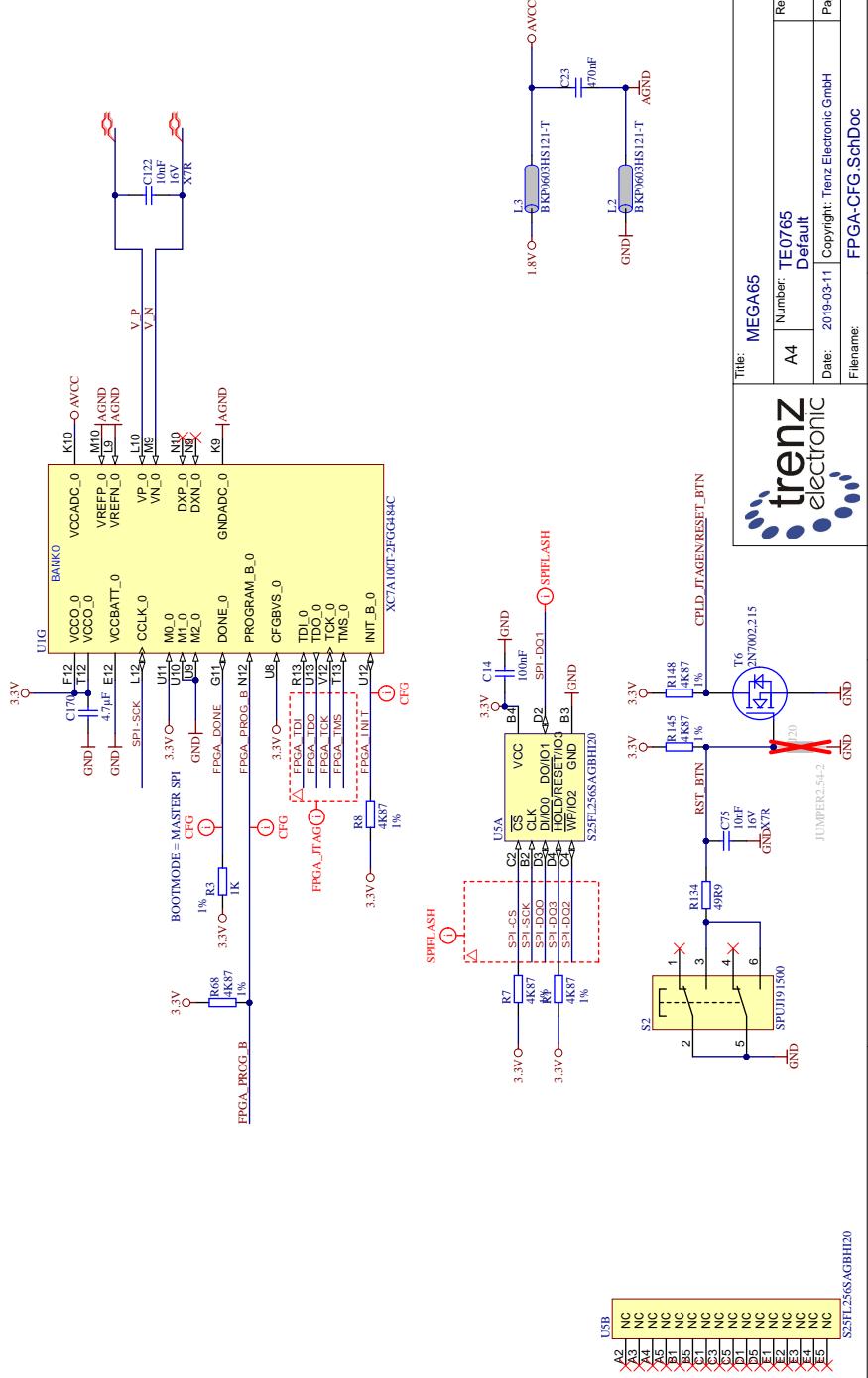
D

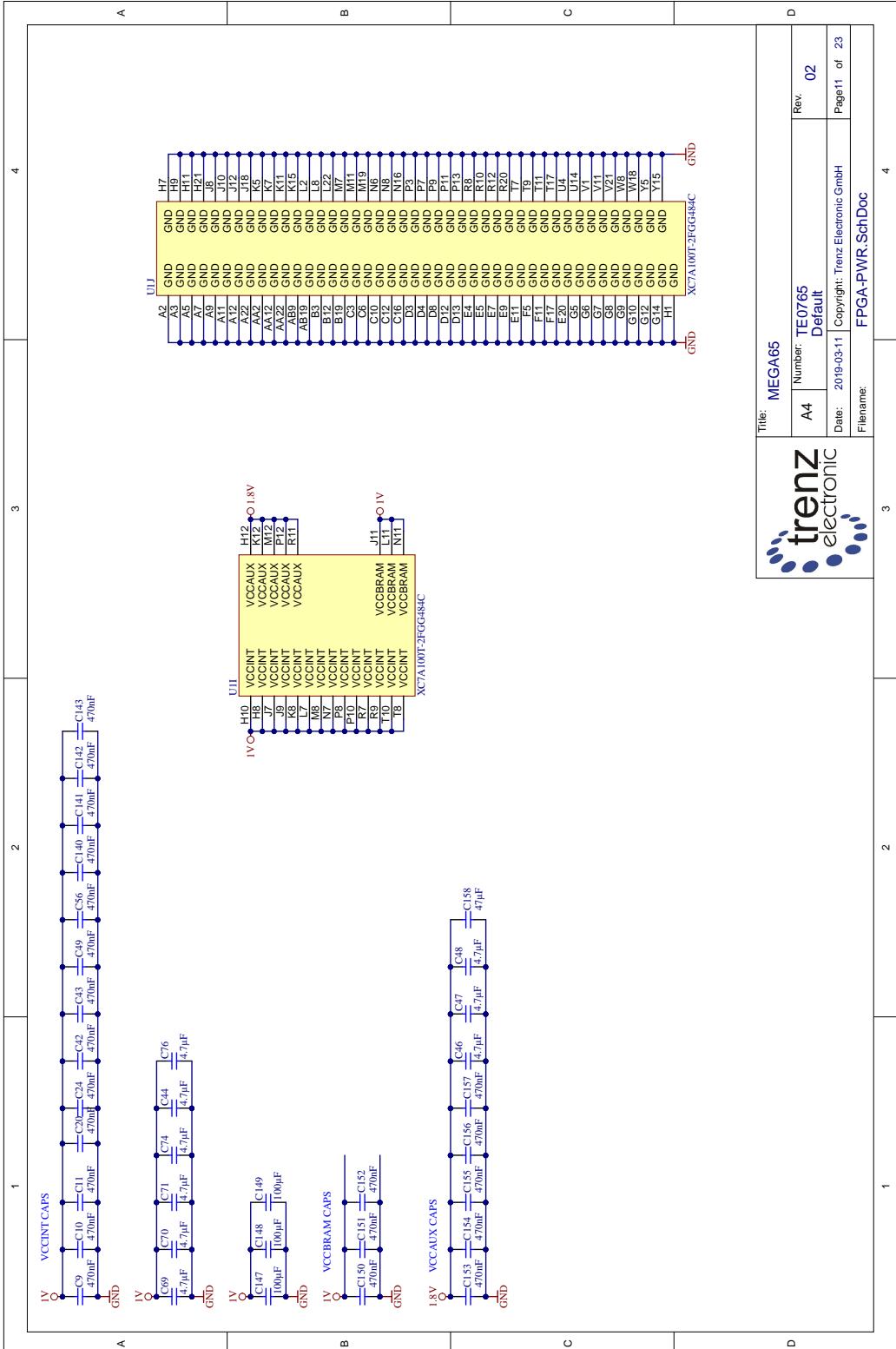
1 2 3 4

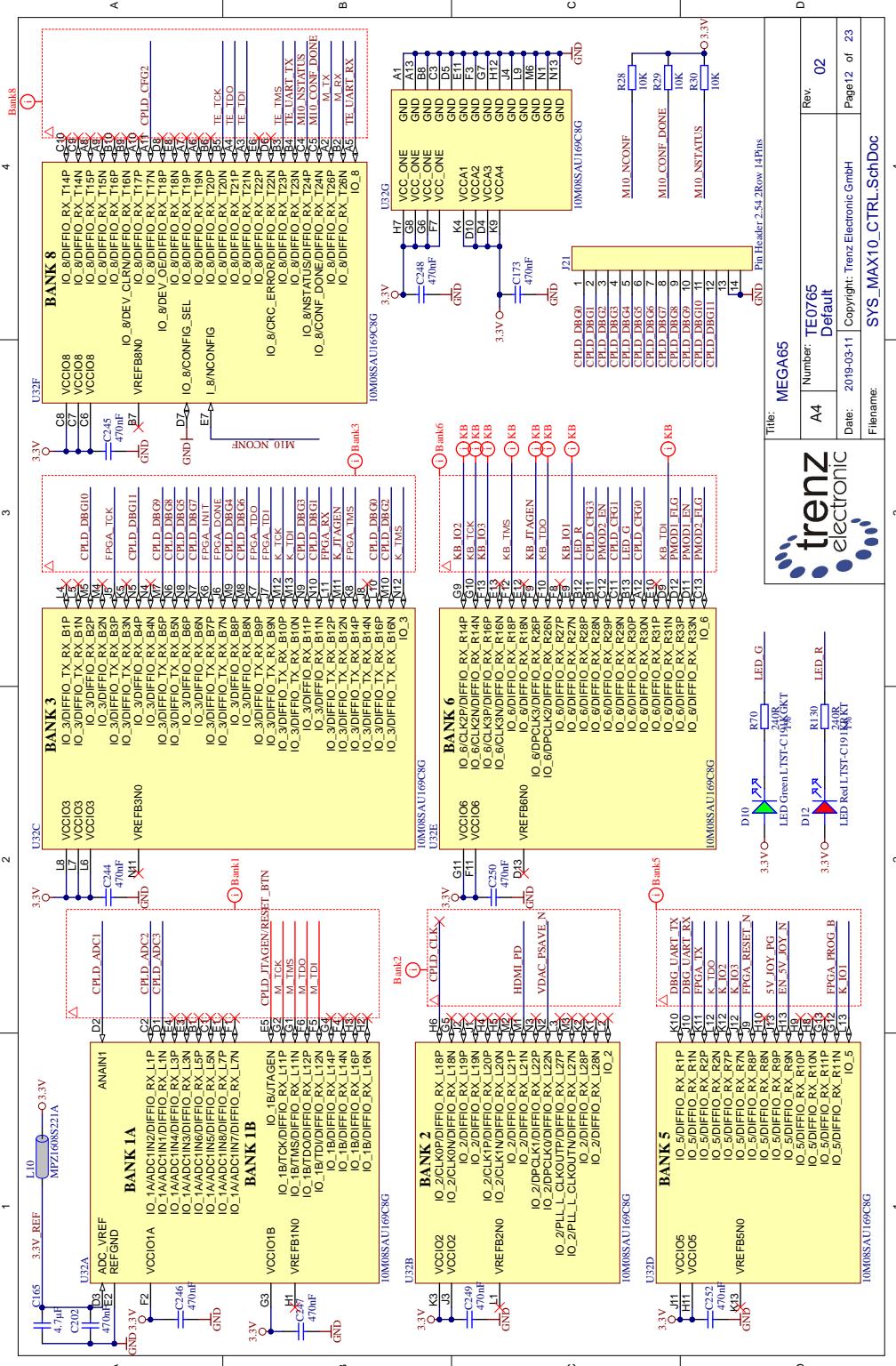


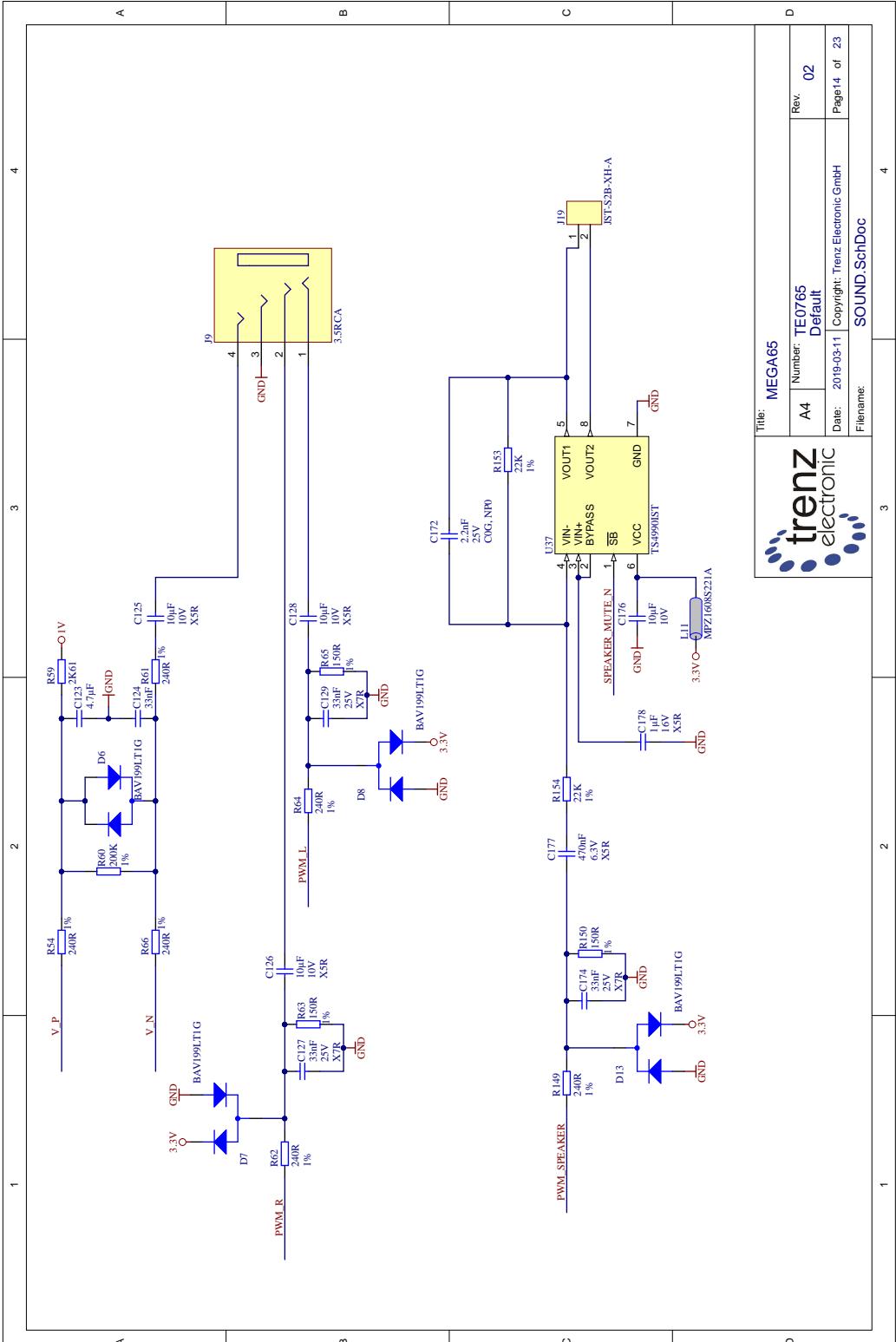
Title: MEGA65	A4 Number: TE0765	Rev. 02
Date: 2019-03-11	Copyright: Trenz Electronic GmbH	Page 7 of 23
Filename: B16.SchDoc		

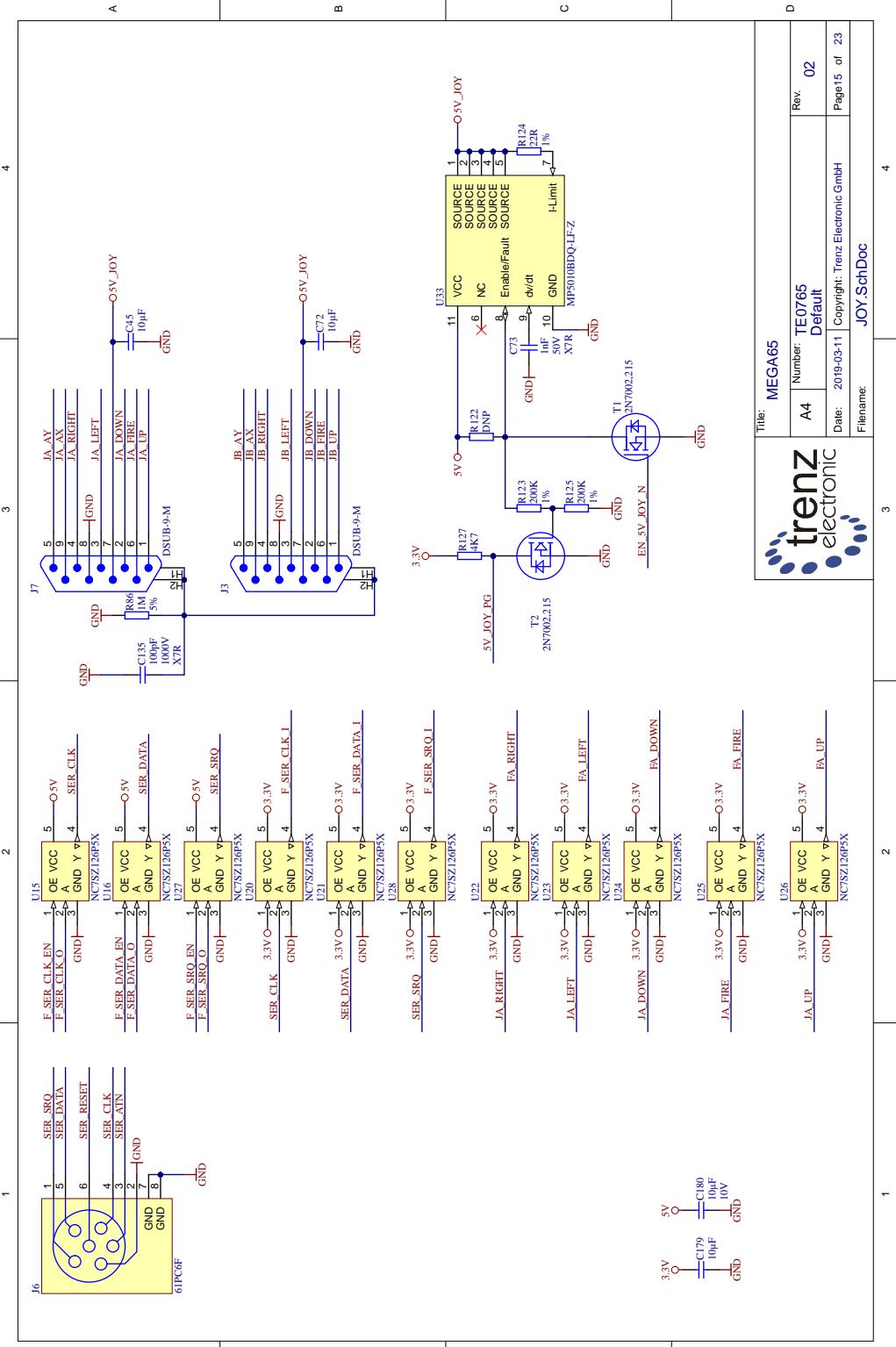












4

3

2

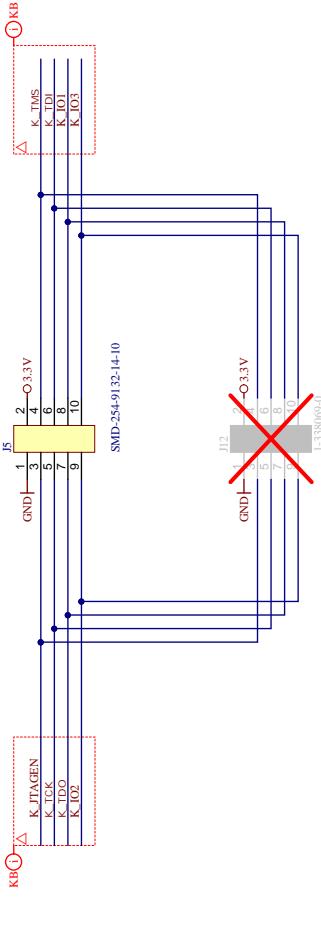
1

A

B

C

D



D

A

B

C

4

3

2

1

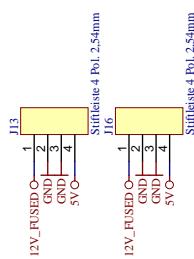
4

3

2

1

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

AA

AB

AC

AD

AE

AF

AG

AH

AI

AJ

AK

AL

AM

AN

AO

AP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

BJ

BK

BL

BM

BN

BO

BP

AQ

AR

AS

AT

AU

AV

AW

AX

AY

AZ

BA

BB

BC

BD

BE

BF

BG

BH

BI

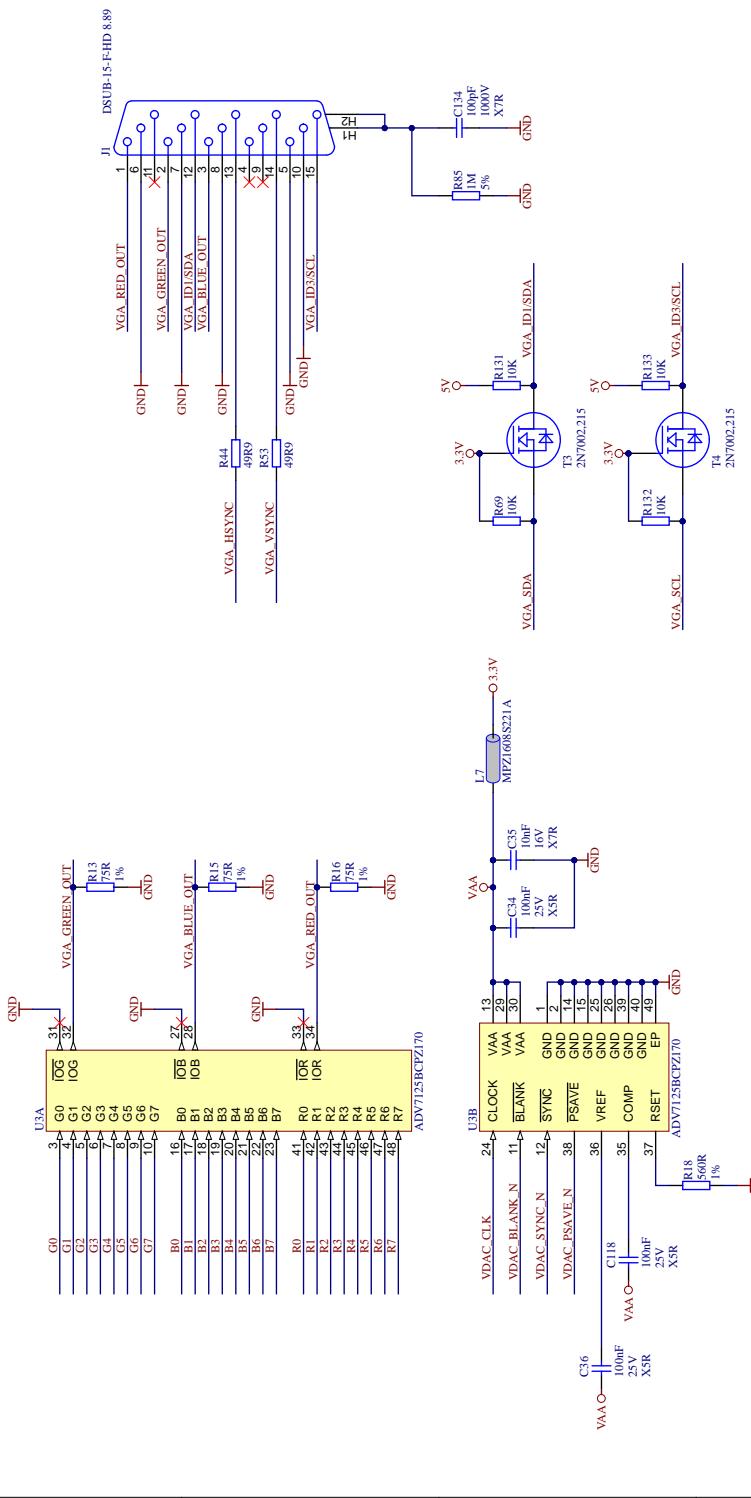
BJ

BK

BL

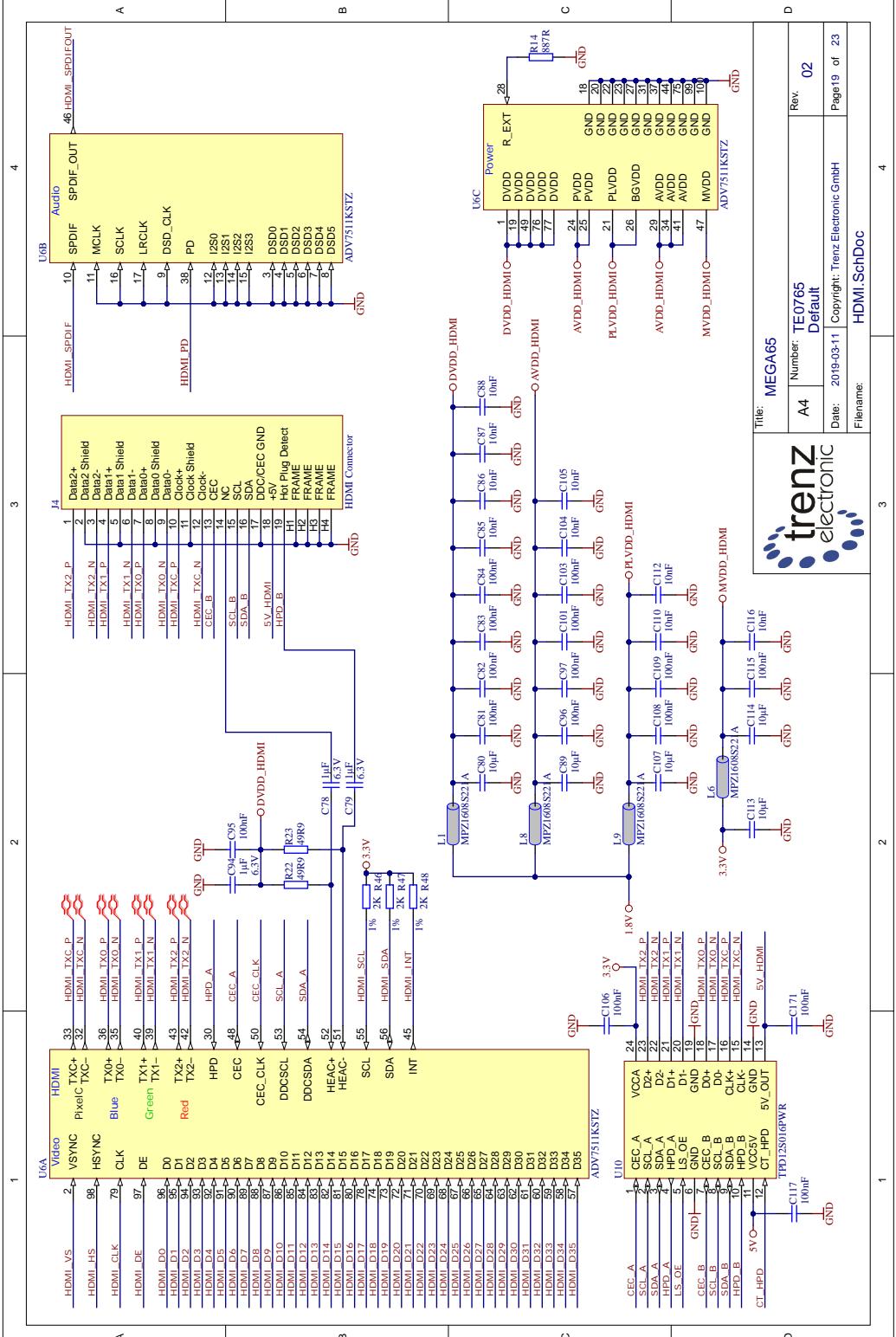
BM

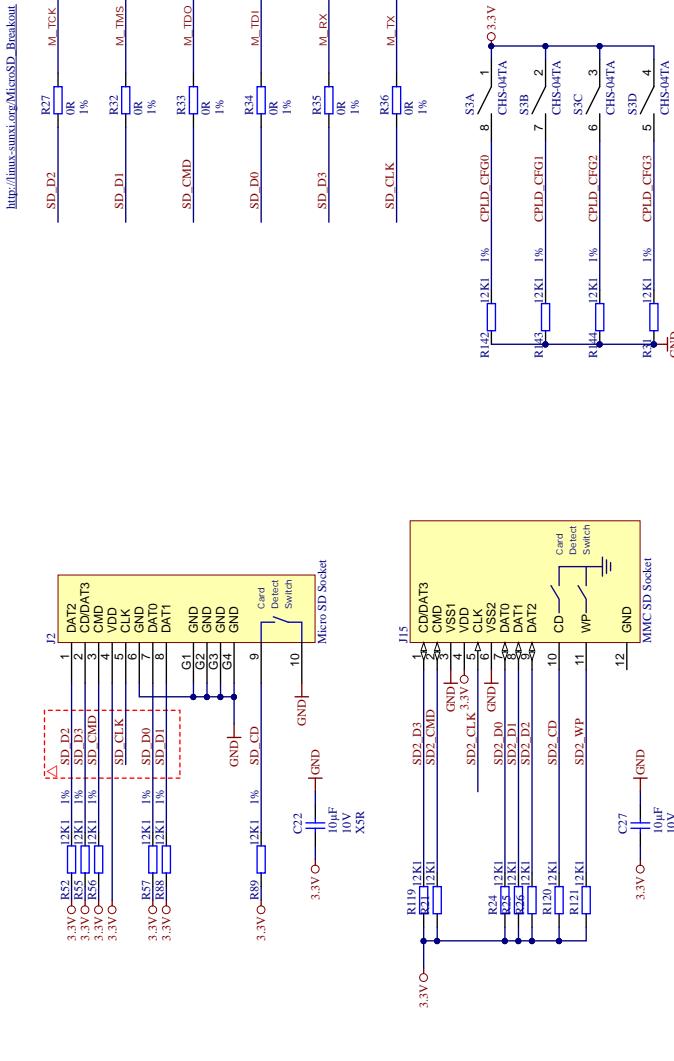
BN



Title:	MEGA65		
A4	Number:	TE0765 Default	Rev. 02
Date:	2019-03-11	Copyright:	Trenz Electronic GmbH
Filename:	VGA_SchDoc		
Z	C	D	Page 18 of 23

The logo for tren electron, featuring the word "tren" in a bold, lowercase sans-serif font above the word "electron" in a smaller, lowercase font. The letters are partially obscured by a circular arrangement of blue ovals.





3: MEGA 65

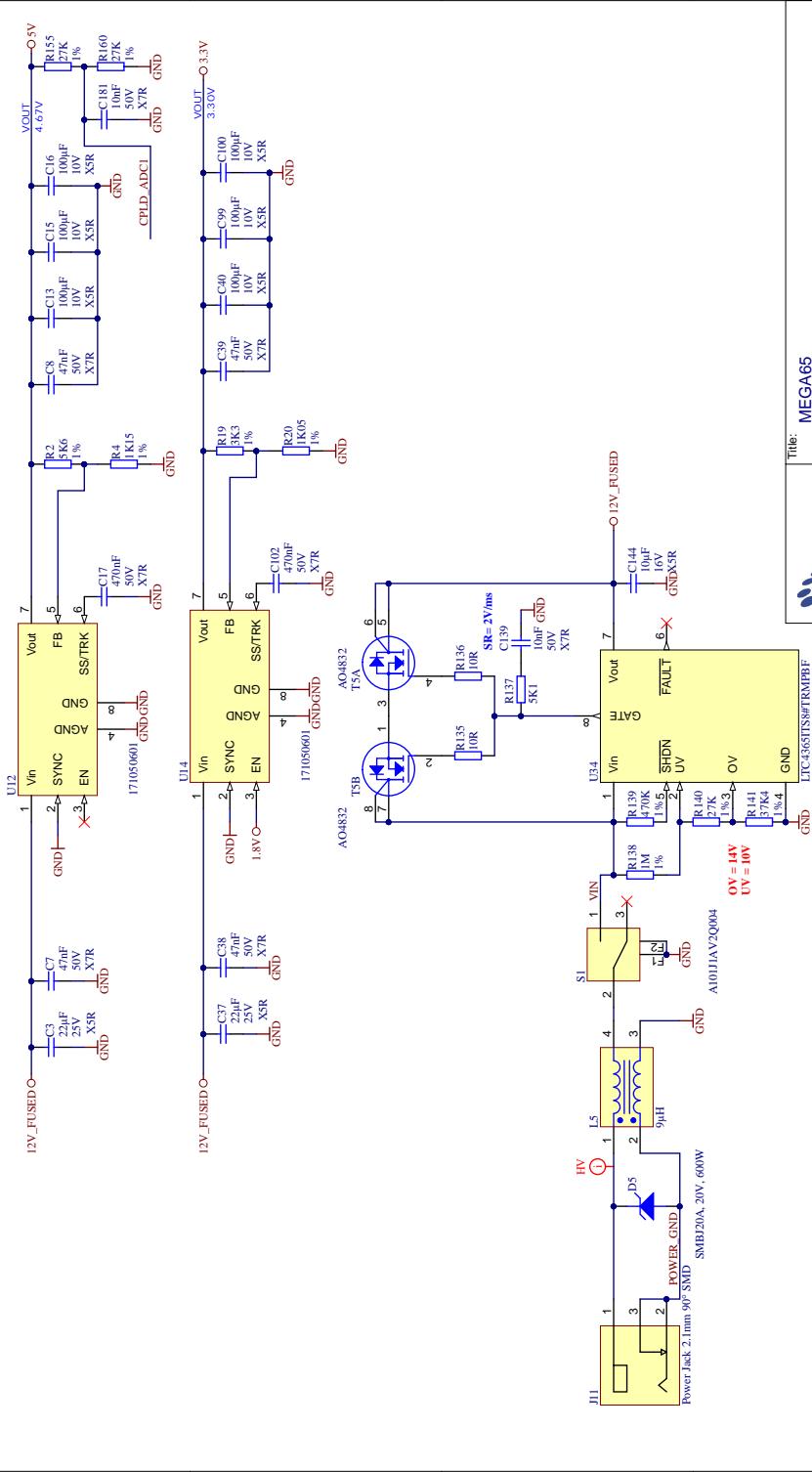
Number:

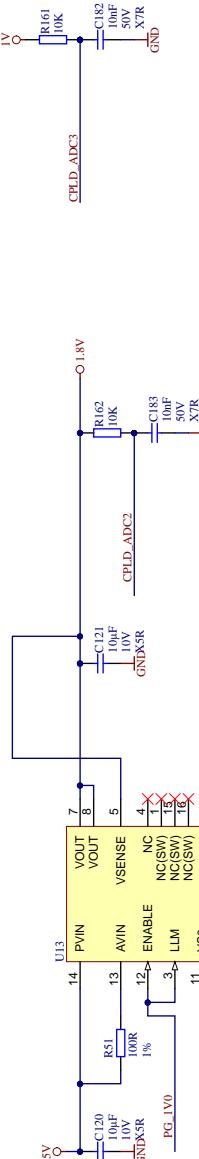
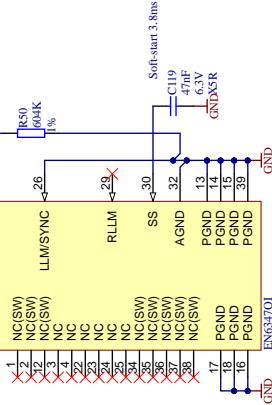
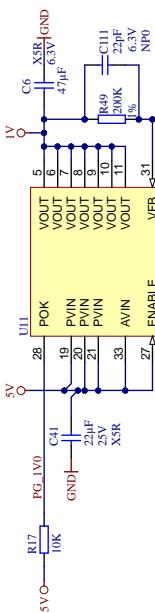
4

e: 2019-03-11



1





tren
electro

MEGA 65

Number:

2019-03-11

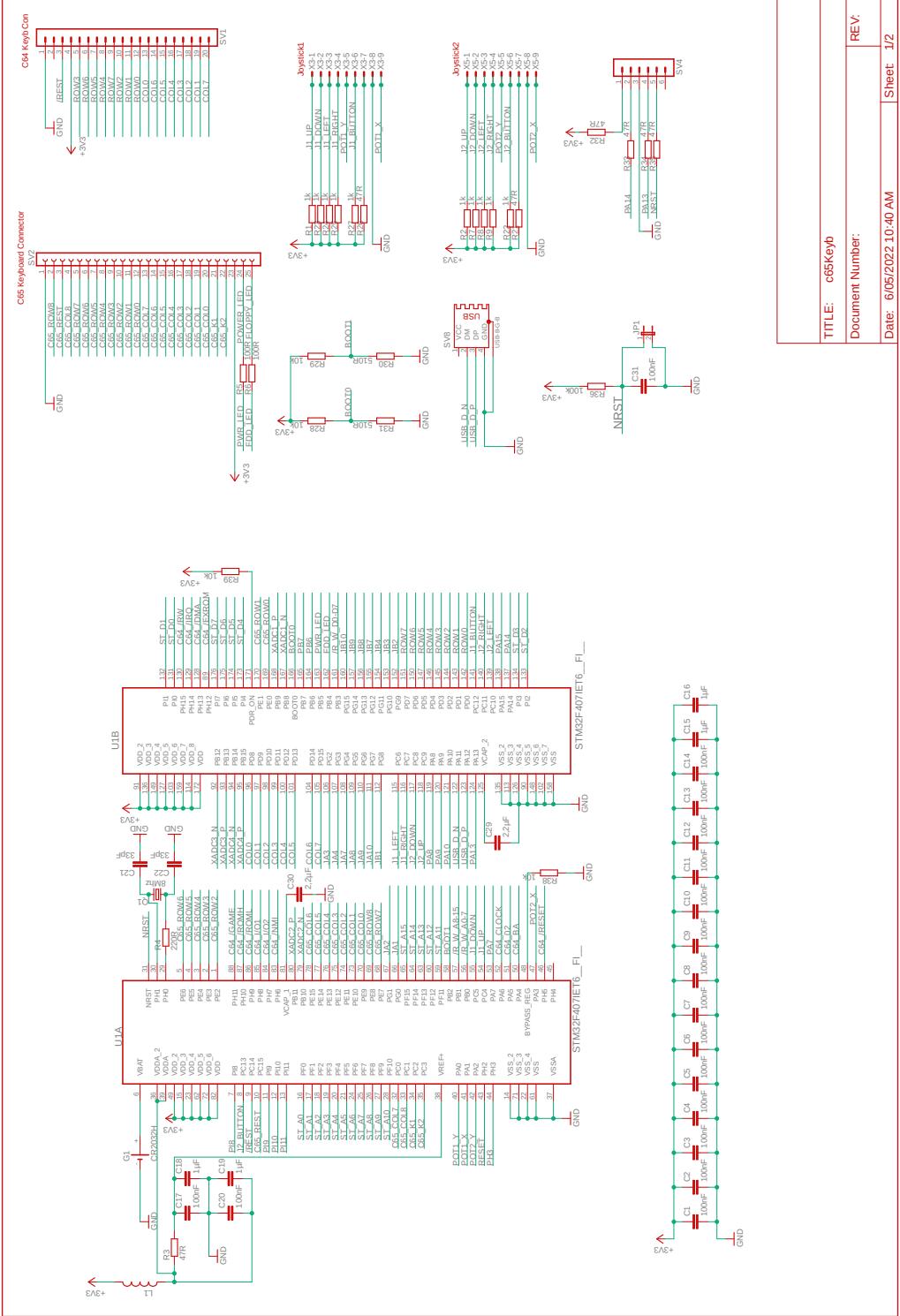
POWER SchDoc

D
Rev. 02

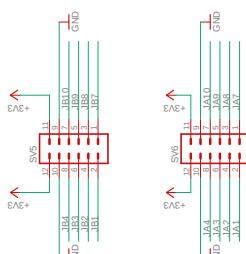
Page 23 of 23

100

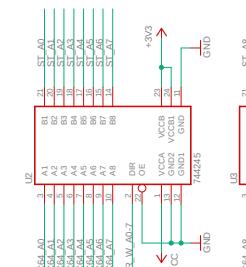
NEXYS WIDGET BOARD SCHEMATICS



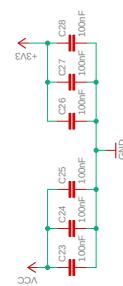
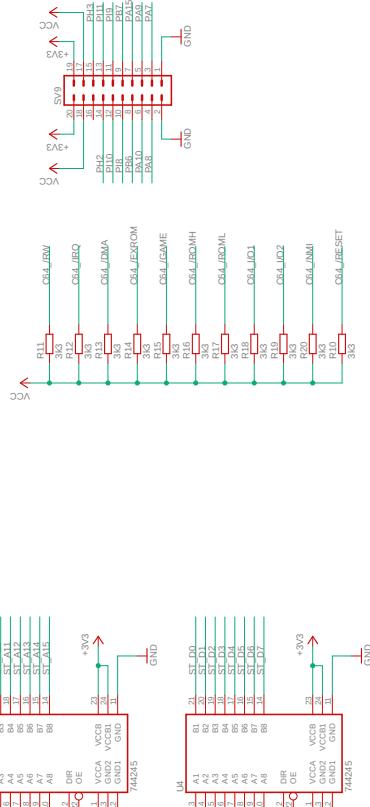
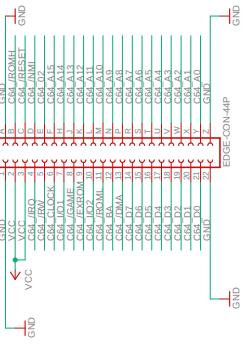
Nexys 4 Input



Output Driver



Expansion Slot

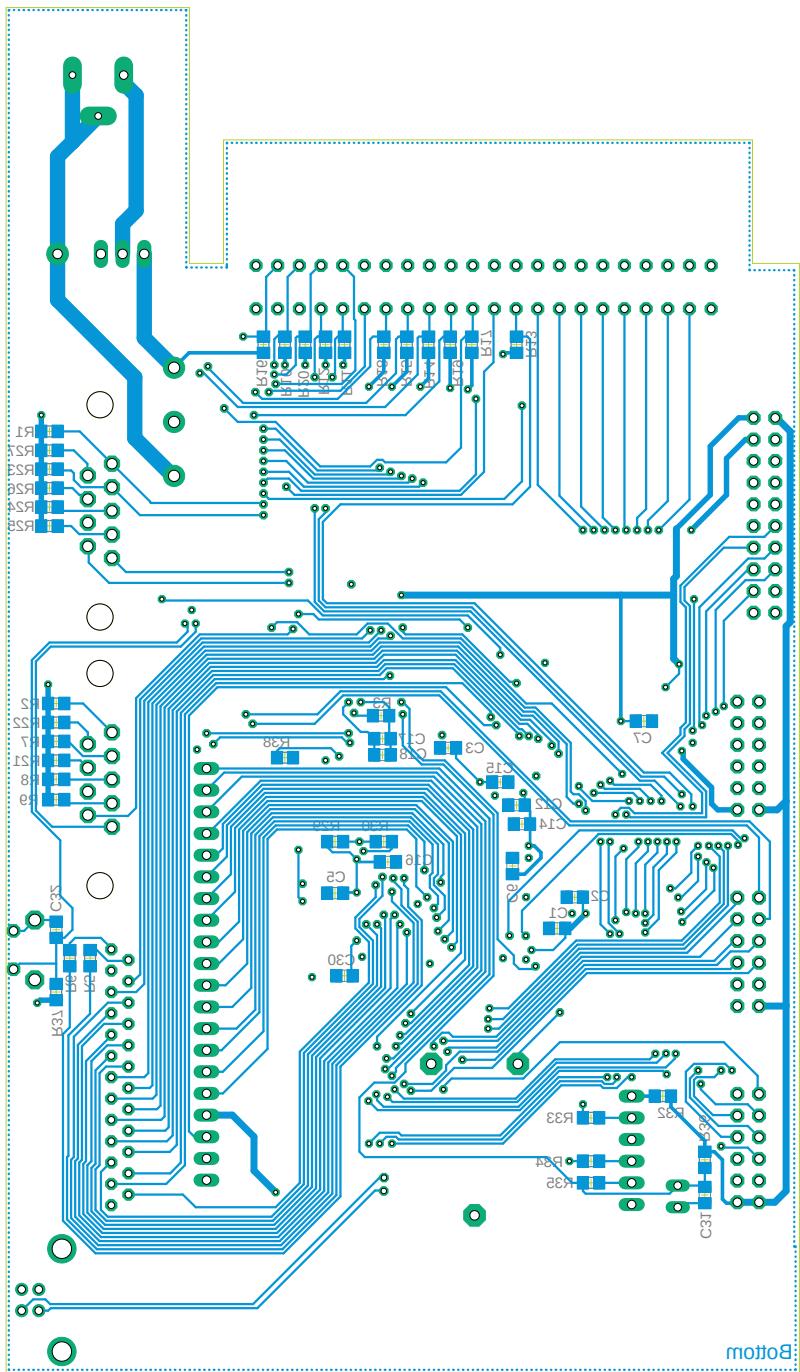


TITLE: c65keyb

Document Number:

Date: 6/05/2022 10:40 AM

REV: 22



APPENDIX X

Supporters & Donors

- **Organisations**
- **Contributors**
- **Supporters**

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

The MEGA Museum of Electronic Games & Art e.V. Germany

EVERYTHING

Trenz Electronik, Germany

MOTHERBOARD

Hintsteiner, Austria

CASE

GMK, Germany

KEYBOARD

CONTRIBUTORS

Andreas Liebeskind

(*libi in paradise*)

CFO MEGA eV

Thomas Hertzler

(*grumpyninja*)

USA Spokesman

Russell Peake

(*rdpeake*)

Bug Herding

Alexander Nik Petra

(*n0d*)

Early Case Design

Ralph Egas

(*O-limits*)

Business Advisor

Lucas Moss

MEGAphone PCB Design

Daren Klamer

(*Impakt*)

Manual proof-reading

Dr. Canan Hastik

(*indica*)

Chairwoman MEGA eV

Simon Jameson

(*Shallan*)

Platform Enhancements

Stephan Kleinert

(*ubik*)

Destroyer of BASIC 10

Wayne Johnson

(*sausage*)

Manual Additions

L. Kleiss

(*LAK132*)

MegaWAT Presentation Software

Maurice van Gils

(*Maurice*)

BASIC 65 example programs

Andrew Owen

(*Cheveron*)

Keyboard, Sinclair Support

SUPPORTERS

@11110110100
3c74ce64
8-Bit Classics
Aaron Smith
Achim Mrotzek
Adolf Nefischer
Adrian Esdaile
Adrien Guichard
Ahmed Kablaoui
Alan Bastian Witkowski
Alan Field
Alastair Paulin-Campbell
Alberto Mercuri
Alexander Haering
Alexander Kaufmann
Alexander Niedermeier
Alexander Soppert
Alfonso Ardire
Amiga On The Lake
André Kudra
André Simeit
André Wösten
Andrea Farolfi
Andrea Minutello
Andreas Behr
Andreas Freier
Andreas Grabski
Andreas Millinger
Andreas Nopper
Andreas Ochs
Andreas Wendel Manufaktur
Andreas Zschunke
Andrew Bingham
Andrew Dixon
Andrew Mondt
Andrzej Hłuchyj
Andrzej Sawiniec
Andrzej Śliwa
Anthony W. Leal
Arkadiusz Bronowicki
Arkadiusz Kwasny
Arnaud Léandre
Arne Drews
Arne Neumann
Arne Richard Tyarks
Axel Klahr
Balaz Ondrej
Barry Thompson
Bartol Filipovic
Benjamin Maas
Bernard Alaiz
Bernhard Zorn
Bieno Marti-Braitmaier
Bigby
Bill LaGrue
Bjoerg Stojalowski
Björn Johannesson
Bjørn Melbøe
Bo Goeran Kvamme
Boerge Noest
Bolko Beutner
Brett Hallen
Brian Gajewski
Brian Green
Brian Juul Nielsen
Brian Reiter
Bryan Pope
Burkhard Franke
Byron Goodman
Cameron Roberton (KONG)
Carl Angervall
Carl Danowski
Carl Stock
Carl Wall
Carlo Pastore
Carlos Silva
Carsten Sørensen
Cenk Miroglu Miroglu
Chang sik Park
Charles A. Hutchins Jr.
Chris Guthrey
Chris Hooper
Chris Stringer
Christian Boettcher
Christian Eick
Christian Gleinser
Christian Gräfe
Christian Heffner
Christian Kersting
Christian Schiller
Christian Streck
Christian Weyer
Christian Wyk
Christoph Haug
Christoph Huck
Christoph Pross
Christopher Christopher
Christopher Kalk
Christopher Kohlert
Christopher Nelson
Christopher Taylor
Christopher Whillock
Claudio Piccinini
Claus Skrepel
Collen Blijenberg
Constantine Lignos
Crnjaninja
Daniel Auger
Daniel Julien
Daniel Lobitz
Daniel O'Connor
Daniel Teicher
Daniel Tootill
Daniel Wedin
Daniele Benetti
Daniele Gaetano Capursi
Dariusz Szczesniak
Darrell Westbury
David Asenjo Raposo
David Dillard
David Gorgon
David Norwood
David Raulo
David Ross
de voughn accooe
Dean Scully
Dennis Jeschke
Dennis Schaffers
Dennis Schierholz

Dennis Schneck	Frank Haaland	Helge Förster
denti	Frank Hempel	Hendrik Fensch
Dick van Ginkel	Frank Koschel	Henning Harperath
Diego Barzon	Frank Linhares	Henri Parfait
Dierk Schneider	Frank Sleeuwaert	Henrik Kühn
Dietmar Krueger	Frank Wolf	Holger Burmester
Dietmar Schinnerl	FranticFreddie	Holger Sturk
Dirk Becker	Fredrik Ramsberg	Howard Knibbs
Dirk Wouters	Fridun Nazaradeh	Hubert de Hollain
Domingo Fivoli	Friedel Kropp	Huberto Kusters
DonChaos	Garrick West	Hugo Maria Gerardus v.d. Aa
Donn Lasher	Gary Lake-Schaal	Humberto Castaneda
Douglas Johnson	Gary Pearson	Ian Cross
Dr. Leopold Winter	Gavin Jones	IDE64 Staff
Dusan Sobotka	Geir Sigmund Straume	Igor Ianov
Earl Woodman	Gerd Mitlaender	Immo Beutler
Ed Reilly	Giampietro Albiero	Ingo Katte
Edoardo Auteri	Giancarlo Valente	Ingo Keck
Eduardo Gallardo	Gianluca Girelli	Insanely Interested Publishing
Eduardo Luis Arana	Giovanni Medina	IT-Dienstleistungen Obsieger
Eirik Juliussen Olsen	Glen Fraser	Ivan Elwood
Emilio Monelli	Glen R Perye III	Jaap HUIJSMAN
EP Technical Services	Glenn Main	Jace Courville
Epic Sound	Gordon Rimac	Jack Wattenhofer
Erasmus Kuhlmann	GRANT BYERS	Jakob Schönplug
ergoGnomik	Grant Louth	Jakub Tyszko
Eric Hilaire	Gregor Bubek	James Hart
Eric Hildebrandt	Gregor Gramlich	James Marshburn
Eric Hill	Guido Ling	James McClanahan
Eric Jutrzenka	Guido von Gösseln	James Sutcliffe
Erwin Reichel	Guillaume Serge	Jan Bitruff
Espen Skog	Gunnar Hemmerling	Jan Hildebrandt
Evangelos Mpouras	Günter Hummel	Jan Iemhoff
Ewan Curtis	Guy Simmons	Jan Kösters
Fabio Zanicotti	Guybrush Threepwood	Jan Peter Borsje
Fabrizio Di Dio	Hakan Blomqvist	Jan Schulze
Fabrizio Lodi	Hans Pronk	Jan Stoltenberg-Lerche
FARA Gießen GmbH	Hans-Jörg Nett	Janne Tompuri
FeralChild	Hans-Martin Zedlitz	Jannis Schulte
First Choice Auto's	Harald Dosch	Jari Loukasmäki
Florian Rienhardt	Harri Salokorpi	Jason Smith
Forum64. de	Harry Culpan	Javier Gonzalez Gonzalez
Francesco Baldassarri	Harry Venema	Jean-Paul Lauque
Frank Fechner	Heath Gallimore	Jeffrey van der Schilden
Frank Glaush	Heinz Roesner	Jens Schneider
Frank Gulasch	Heinz Stampfli	Jens-Uwe Wessling

Jesse DiSimone	Kevin Edwards	Marco Rivelas
Jett Adams	Kevin Thomasson	Marco van de Water
Johan Arneklev	Kim Jorgensen	Marcus Gerards
Johan Berntsson	Kim Rene Jensen	Marcus Herbert
Johan Svensson	Kimmo Hamalainen	Marcus Linkert
Johannes Fitz	Konrad Buryło	Marek Pernicky
John Cook	Kosmas Einbrodt	Mario Esposito
John Deane	Kurt Klemm	Mario Fetka
John Dupuis	Lachlan Glaskin	Mario Teschke
John Nagi	Large bits collider	Mariusz Tymków
John Rorland	Lars Becker	Mark Adams
John Sargeant	Lars Edelmann	Mark Anderson
John Traeholt	Lars Slivsgaard	Mark Green
Jon Sandelin	Lasse Lambrecht	Mark Hucker
Jonas Bernemann	Lau Olivier	Mark Leitiger
Jonathan Prosise	Lee Chatt	Mark Spezzano
Joost Honig	Loan Leray	Mark Watkin
Jordi Pakey-Rodriguez	Lorenzo Quadri	Marko Rizvic
Jöre Weber	Lorenzo Travagli	Markus Bieler
Jörg Jungermann	Lorin Millsap	Markus Bonet
Jörg Schaeffer	Lothar James Foss	Markus Dauberschmidt
Jörg Weese	Lothar Serra Mari	Markus Fehr
Josef Hesse	Luca Papinutti	Markus Fuchs
Josef Soucek	Ludek Smetana	Markus Guenther-Hirn
Josef Stohwasser	Lukas Burger	Markus Liukka
Joseph Clifford	Lutz-Peter Buchholz	Markus Merz
Joseph Gerth	Luuk Spaetgens	Markus Roesgen
Jovan Crnjainin	Mad Web Skills	Markus Uttenweiler
Juan Pablo Schisano	MaDCz	Martin Bauhuber
Juan S. Cardona Iguina	Magnus Wiklander	Martin Benke
JudgeBeeb	Maik Diekmann	Martin Gendera
Juliusen Olsen	Malte Mundt	Martin Groß
Juna Luis Fernandez Garcia	Manfred Wittemann	Martin Gutenbrunner
Jürgen Endras	Manuel Beckmann	Martin Johansen
Jürgen Herm Stapelberg	Manzano Mérida	Martin Marbach
Jyrki Laurila	Marc "3D-vice" Schmitt	Martin Sonleitner
Kai Pernau	Marc Bartel	Martin Steffen
Kalle Pöyhönen	Marc Jensen	Marvin Hardy
Karl Lamford	Marc Schmidt	Massimo Villani
Karl-Heinz Blum	Marc Theunissen	Mathias Dellacherie
Karsten Engstler	Marc Tutor	Mathieu Chouinard
Karsten Westebbe	Marc Wink	Matthew Adams
katarakt	Marcel Buchtmann	Matthew Browne
Keith McComb	Marcel Kante	Matthew Carnevale
Kenneth Dyke	Marco Beckers	Matthew Palmer
Kenneth Joensson	Marco Cappellari	Matthew Santos

Matthias Barthel	Miguel Angel Rodriguez Jodar	Paul Johnson
Matthias Dolenc	Mikael Lund	Paul Kuhnast (mindrail)
Matthias Fischer	Mike Betz	Paul Massay
Matthias Frey	Mike Kastrantas	Paul Westlake
Matthias Grandis	Mike Pikowski	Paul Woegerer
Matthias Guth	Mikko Hämäläinen	Pauline Brasch
Matthias Lampe	Mikko Suontausta	Paulo Apolonia
Matthias Meier	Mirko Roller	Pete Collin
Matthias Mueller	Miroslav Karkus	Pete of Retrohax.net
Matthias Nofer	Morgan Antonsson	Peter Eliades
Matthias Schonder	Moritz	Peter Gries
Maurice Al-Khaliedy	Morten Nielsen	Peter Habura
Max Ihlenfeldt	MUBIQUO APPS,SL	Peter Herklotz
Meeso Kim	Myles Cameron-Smith	Peter Huyoff
Michael Dailly	Neil Moore	Peter Knörzer
Michael Dötsch	Nelson	Peter Leswell
Michael Dreßel	neoman	Peter Weile
Michael Fichtner	Nicholas Melnick	Petri Alvinen
Michael Fong	Nikolaj Brinch Jørgensen	Philip Marien
Michael Geoffrey Stone	Nils Andreas	Philip Timmermann
Michael Gertner	Nils Eilers	Philipp Rudin
Michael Grün	Nils Hammerich	Pierre Kressmann
Michael Habel	Nils77	Pieter Labie
Michael Härtig	Norah Smith	Piotr Kmiecik
Michael Haynes	Norman King	Power-on.at
Michael J Burkett	Normen Zoch	Przemysław Safonow
Michael Jensen	Olaf Grunert	Que Labs
Michael Jurisch	Ole Eitels	R Welbourn
Michael Kappelgaard	Oliver Boerner	R-Flux
Michael Kleinschmidt	Oliver Brüggemann	Rafał Michno
Michael Lorenz	Oliver Graf	Rainer Kappler
Michael Mayerhofer	Oliver Smith	Rainer Kopp
Michael Nurney	Olivier Bori	Rainer Weninger
Michael Rasmussen	ONEPSI LLC	Ralf Griewel
Michael Richmond	oRdYNe	Ralf Pöscha
Michael Sachse	Osaühing Trioflex	Ralf Reinhardt
Michael Sarbak	OSHA-PROS USA	Ralf Schenden
Michael Schneider	Padawer	Ralf Smolarek
Michael Scholz	Patrick Becher	Ralf Zenker
Michael Timm	Patrick Bürkstümmer	Ralph Bauer
Michael Traynor	Patrick de Zoete	Ralph Wernecke
Michael Whipp	Patrick Toal	Rédl Károly
Michal Ursiny	Patrick Vogt	Reiner Lanowski
Michele Chiti	Paul Alexander Warren	Remi Veilleux
Michele Perini	Paul Gerhardt (KONG)	Riccardo Bianchi
Michele Porcu	Paul Jackson	Richard Englert

Richard Good
Richard Menedetter
Richard Sopuch
Rick Reynolds
Rico Gruninger
Rob Dean
Robert Bernardo
Robert Eaglestone
Robert Grasböck
Robert Miles
Robert Schwan
Robert Shively
Robert Tangmar
Robert Trangmar
Rodney Xerri
Roger Olsen
Roger Pugh
Roland Attila Kett
Roland Evers
Roland Schatz
Rolf Hass
Ronald Cooper
Ronald Hunn
Ronny Hamida
Ronny Preiß
Roy van Zundert
Rüdiger Wohlfomm
Ruediger Schlenter
Rutger Willemsen
Sampo Peltonen
Sarmad Gilani
SAS74
Sascha Hesse
Scott Halman
Scott Hollier
Scott Robison
Sebastian Baranski
Sebastian Bölling
Sebastian Felzmann
Sebastian Lipp
Sebastian Rakel
Şemseddin Moldibi
Seth Morabito
Shawn McKee
Siegfried Hartmann
Sigurbjörn Larusson
Sigurdur Finnsson
Simon Lawrence
Simon Wolf
spreen.digital
Stefan Haberl
Stefan Kramperth
Stefan Richter
Stefan Schultze
Stefan Sonnek
Stefan Theil
Stefan Vrampe
Stefano Canali
Stefano Mozzi
Steffen Reiersen
Stephan Bielmann
Stephen Jones
Stephen Kew
Steve Gray
Steve Kurlin
Steve Lemieux
Steven Combs
Stewart Dunn
Stuart Marsh
Sven Neumann
Sven Stache
Sven Sternberger
Sven Wiegand
Szabolcs Bence
Tantrumedia Limited
Techvana Operations Ltd.
Teddy Turmeaux
Teemu Korvenpää
The Games Foundation
Thierry Supplisson
Thieu-Duy Thai
Thomas Bierschenk
Thomas Edmister
Thomas Frauenknecht
Thomas Gitzen
Thomas Gruber
Thomas Haidler
Thomas Jager
Thomas Karlßen
Thomas Laskowski
Thomas Marschall
Thomas Niemann
Thomas Scheelen
Thomas Schilling
Thomas Tahsin-Bey
Thomas Walter
Thomas Wirtzmann
Thorsten Knoll
Thorsten Nolte
Tim Krome
Tim Waite
Timo Weirich
Timothy Blanks
Timothy Henson
Timothy Prater
Tobias Butter
Tobias Heim
Tobias Köck
Tobias Lüthi
Tommi Vasarainen
Toni Ammer
Tore Olsen
Torleif Strand
Torsten Schröder
Tuan Nguyen
Uffe Jakobsen
Ulrich Hintermeier
Ulrich Nieland
Ulrik Kruse
Ursula Förstle
Uwe Anfang
Uwe Boschanski
Vedran Vrbanc
Verm Project
Wayne Rittmann
Wayne Sander
Wayne Steele
Who Knows
Winfried Falkenhahn
Wolfgang Becker
Wolfgang Stabla
Worblehat
www.patop69.net
Yan B
Zoltan Markus
Zsolt Zsila
Zytex Online Store

Bibliography

- [1] N. Montfort, P. Baudoin, J. Bell, I. Bogost, J. Douglass, M. C. Marino, M. Mateas, C. Reas, M. Sample, and N. Vawter, *10 PRINT CHR \$(205.5+ RND (1));: GOTO 10*. MIT Press, 2012.
- [2] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exceptionless system calls." in *Osdi*, vol. 10, 2010, pp. 1-8.
- [3] Actraiser, "Vic-ii for beginners: Screen modes, cheaper by the dozen," 2013. [Online]. Available: <http://dustlayer.com/vic-ii/2013/4/26/vic-ii-for-beginners-screen-modes-cheaper-by-the-dozen>

INDEX

, (comma), [8-22](#), [8-39](#)
: (colon), [8-22](#), [8-39](#), [8-40](#)
<> (not equal to), [8-31](#)
\$00 (PORTDDR), [G-16](#)
\$00 (STOPTX), [Q-11](#)
\$01 (PORT), [G-16](#)
\$01 (STARTTX), [Q-11](#)
\$D0 (RXNORMAL), [Q-11](#)
\$D4 (DEBUGVIC), [Q-11](#)
\$DC (DEBUGCPU), [Q-11](#)
\$DE (RXONLYONE), [Q-11](#)
\$F1 (FRAME1K), [Q-11](#)
\$F2 (FRAME2K), [Q-11](#)

ADC, [H-18](#), [H-64](#), [H-125](#)
ADCQ, [G-14](#), [H-126](#)
ALR, [H-18](#)
altpal, [17-8](#)
Amiga™ style audio, [L-20](#)
ANC, [H-19](#)
AND, [8-44](#), [H-20](#), [H-64](#), [H-127](#)
ANDQ, [G-14](#), [H-128](#)
ARR, [H-20](#)
ASL, [H-21](#), [H-65](#)
ASLQ, [G-14](#), [H-128](#)
ASR, [H-66](#)
ASRQ, [G-14](#), [H-129](#)
ASSEMBLE, [K-6](#)
ASW, [H-67](#)
audio cross-bar switch, [R-23](#)
audio mixer, [R-23](#)

BACKGROUND, [8-44](#)
BASIC 65 Arrays, [B-7](#)
BASIC 65 Commands, [B-240](#)
 APPEND, [B-20](#)
 AUTO, [B-23](#)
 BACKGROUND, [8-44](#), [B-24](#)
 BACKUP, [B-26](#)
 BANK, [B-27](#), [F-4](#)
 BEGIN, [B-28](#)
 BEND, [B-29](#)
 BLOAD, [B-30](#)

BOOT, [B-32](#)
BORDER, [8-44](#), [B-33](#)
BOX, [B-34](#)
BSAVE, [B-36](#)
BUMP, [B-38](#)
BVERIFY, [B-39](#)
CATALOG, [B-40](#)
CHANGE, [B-42](#)
CHAR, [B-43](#)
CHARDEF, [B-45](#)
CHDIR, [B-46](#)
CIRCLE, [B-48](#)
CLOSE, [B-51](#)
CLR, [B-52](#)
CMD, [B-54](#)
COLLECT, [B-55](#)
COLLISION, [B-56](#)
COLOR, [B-57](#)
CONCAT, [B-58](#)
CONT, [8-39](#), [B-59](#)
COPY, [B-60](#)
CURSOR, [B-63](#)
CUT, [B-64](#)
DATA, [B-65](#)
DCLEAR, [B-66](#)
DCLOSE, [B-67](#)
DEF FN, [B-69](#)
DELETE, [B-70](#)
DIM, [B-71](#)
DIR, [B-72](#)
Direct Mode, [B-5](#)
DISK, [B-74](#)
DLOAD, [B-75](#)
DMA, [B-77](#), [F-4](#)
DMODE, [B-78](#)
DO, [B-79](#)
DOT, [B-82](#)
DPAT, [B-83](#)
DSAVE, [B-86](#)
DVERIFY, [B-88](#)
EDMA, [B-91](#)
ELLIPSE, [B-93](#)

ELSE, B-96
END, 8-39, B-98
ENVELOPE, B-99
ERASE, B-101
EXIT, B-103
FAST, B-105
FGOSUB, B-106
FGOTO, B-107
FILTER, B-108
FIND, B-109
FONT, B-111
FOR, 8-7, B-112
FOREGROUND, 8-44, B-113
FORMAT, B-114
FREEZER, B-117
GCOPY, B-119
GET, B-120
GET#, B-121
GETKEY, B-122
GO64, B-123
GOSUB, B-124
GOTO, 8-46, B-125
GRAPHIC, B-126
HEADER, B-127
HELP, B-128
HIGHLIGHT, B-130
IF, 8-30, B-131
IMPORT, B-132
INPUT, 8-17, 8-24, B-133
INPUT#, B-134
INSTR, B-136
KEY, B-139
LET, 8-15, B-143
LINE, B-144
LINE INPUT#, B-145
LIST, 8-11, 8-12, B-146
LOAD, B-147
LOADIFF, B-149
LOCK, B-151
LOOP, B-154
MERGE, B-157
MKDIR, B-159
MONITOR, B-161
MOUNT, B-162
MOUSE, B-163
MOVSPR, B-164
NEW, B-166
NEXT, B-167
OFF, B-169
ON, B-170
OPEN, B-172
PAINT, B-174
PALETTE, B-175
PASTE, B-177
PEN, B-179
PLAY, B-181
POLYGON, B-186
PRINT, 8-11, B-189
PRINT USING, B-191
PRINT#, B-190
RCURSOR, B-194
READ, B-195
RECORD, B-196
REM, B-198
RENAME, B-199
RENUMBER, 8-38, B-200
RESTORE, B-202
RESUME, B-203
RETURN, B-204
RMOUSE, B-207
RREG, B-212
RUN, B-217
SAVE, B-219
SAVEIFF, B-220
SCNCLR, B-221
SCRATCH, B-222
SCREEN, B-223
SET, B-226
SLEEP, B-230
SOUND, B-231
SPEED, B-233
SPRCOLOR, B-234
SPRITE, B-235
SPRSAV, B-236

STEP, [B-13](#), [B-239](#)
STOP, [8-39](#)
SYS, [B-242](#)
TEMPO, [B-246](#)
THEN, [8-30](#), [B-247](#)
TO, [B-250](#)
TRAP, [B-251](#)
TROFF, [B-252](#)
TRON, [B-253](#)
TYPE, [B-254](#)
UNLOCK, [B-255](#)
UNTIL, [B-256](#)
USING, [B-257](#)
VERIFY, [B-261](#)
VIEWPORT, [B-262](#)
VOL, [B-263](#)
VSYNC, [B-264](#)
WAIT, [B-265](#)
WHILE, [B-266](#)
WINDOW, [B-267](#)

BASIC 65 Constants, [B-7](#)

BASIC 65 Examples

- ABS, [B-18](#), [B-79](#), [B-154](#), [B-228](#),
[B-256](#), [B-266](#)
- AND, [B-19](#), [B-28](#), [B-38](#), [B-114](#),
[B-127](#), [B-133](#), [B-135](#),
[B-138](#), [B-168](#), [B-171](#),
[B-188](#), [B-197](#), [B-231](#),
[B-235](#), [B-249](#), [B-259](#)
- BACKGROUND, [6-8](#)
- BANK, [10-7](#)
- BOOT, [B-32](#)
- BORDER, [6-8](#)
- BUMP, [B-38](#), [B-56](#)
- CHAR, [B-44](#), [B-95](#)
- COLLISION, [B-56](#)
- COLOR, [B-169](#), [B-171](#), [B-175](#)
- CONCAT, [B-58](#)
- CONT, [B-59](#)
- DIM, [B-7](#), [B-71](#), [B-112](#), [B-116](#),
[B-118](#), [B-133](#), [B-135](#),
- DO, [B-79](#), [B-103](#), [B-120](#), [B-121](#),
[B-124](#), [B-125](#), [B-133](#),
[B-154](#), [B-235](#), [B-256](#),
[B-266](#)
- ELLIPSE, [B-95](#)
- ELSE, [B-97](#), [B-108](#), [B-131](#),
[B-231](#), [B-247](#)
- END, [B-56](#), [B-65](#), [B-71](#), [B-92](#),
[B-97](#), [B-98](#), [B-100](#), [B-102](#),
[B-107](#), [B-131](#), [B-203](#),
[B-204](#), [B-214](#), [B-244](#),
[B-247](#), [B-251](#), [B-264](#)
- ERR, [B-92](#), [B-100](#), [B-102](#), [B-203](#),
[B-251](#)
- FAST, [B-105](#)
- FGOSUB, [B-106](#)
- FGOTO, [B-107](#)
- FOR, [B-56](#), [B-65](#), [B-69](#), [B-71](#),
[B-108](#), [B-110](#)-[B-112](#),
[B-135](#), [B-145](#), [B-160](#),
[B-167](#), [B-184](#), [B-189](#),
[B-190](#), [B-195](#), [B-197](#),
[B-202](#)-[B-205](#), [B-208](#),
[B-232](#), [B-239](#), [B-244](#),
[B-246](#), [B-248](#)-[B-253](#),
[B-263](#), [B-270](#)
- GCOPY, [B-119](#)
- GET, [B-28](#), [B-79](#), [B-103](#), [B-120](#),
[B-121](#), [B-154](#), [B-256](#),
[B-266](#)
- GO64, [10-3](#)
- IMPORT, [B-132](#)
- INPUT, [B-96](#), [B-103](#), [B-106](#),
[B-107](#), [B-121](#), [B-124](#),
[B-125](#), [B-131](#), [B-133](#),
[B-135](#), [B-145](#), [B-197](#),
[B-238](#), [B-247](#)
- JOY, [B-138](#), [B-171](#)

KEY, B-4, B-140, B-169, B-171
LEN, B-28, B-79, B-142, B-154,
B-256, B-266
LINE, B-103, B-121, B-126,
B-144, B-145, B-174,
B-176, B-179, B-205,
B-220, B-221, B-225,
B-238
LOAD, 6-8, B-140, B-148,
B-149, B-235, B-236
LOG, B-104, B-152
LOOP, B-79, B-103, B-120,
B-121, B-124, B-125,
B-133, B-154, B-256,
B-266
MAP, 10-8
MONITOR, B-161
MOUNT, B-162
MOVSPR, B-56, B-165, B-235
NEW, B-166
ON, B-63, B-134, B-138, B-140,
B-148, B-149, B-163,
B-171, B-194, B-204,
B-207, B-226, B-228
OPEN, B-51, B-54, B-103,
B-121, B-126, B-150,
B-172, B-176, B-179,
B-205, B-210, B-221,
B-225
PEEK, 10-7
PEN, B-50, B-64, B-126, B-144,
B-174, B-176, B-177,
B-179, B-205, B-210,
B-220, B-221, B-225,
B-262
POKE, 3-11, 10-5, 10-7
POLYGON, B-186
POT, B-188
PRINT, B-4, B-5, B-18, B-19,
B-21, B-22, B-28, B-29,
B-38, B-47, B-51, B-52,
B-54, B-56, B-59, B-62,
B-63, B-65, B-68, B-69,
B-71, B-85, B-87, B-92,
B-96-B-98, B-100,
B-102-B-104,
B-106-B-108,
B-110-B-112, B-115,
B-121, B-124, B-125,
B-128, B-129, B-131,
B-133, B-135, B-138,
B-141, B-142, B-145,
B-152, B-153, B-155,
B-158, B-160, B-167,
B-168, B-171, B-173,
B-178, B-184, B-187,
B-189, B-190, B-192,
B-194, B-195, B-197,
B-202-B-210, B-212,
B-214, B-218, B-229,
B-232, B-237-B-241,
B-244, B-245,
B-247-B-253,
B-258-B-260, B-268,
B-270
RCURSOR, B-194
RECORD, B-197
REM, B-25-B-29, B-31, B-33,
B-38, B-45-B-47, B-50,
B-51, B-53, B-54, B-56,
B-61, B-63-B-65, B-67,
B-69-B-71, B-74, B-76,
B-77, B-79, B-81, B-91,
B-92, B-95, B-96, B-98,
B-100-B-103,
B-105-B-108, B-110,
B-111, B-117,
B-119-B-122,
B-124-B-126, B-131,
B-133, B-135-B-138,
B-140, B-143-B-146,
B-148-B-151,
B-154-B-156, B-159,
B-162, B-163, B-166,

B-169, B-171, B-172,
B-174-B-179,
B-183-B-190, B-193,
B-197-B-199, B-201,
B-204, B-205,
B-207-B-211,
B-213-B-218,
B-220-B-222,
B-225-B-228,
B-230-B-234, B-236,
B-238, B-240, B-243,
B-247-B-250, B-252,
B-253, B-255, B-256,
B-259, B-262,
B-264-B-269
RIGHT, B-206
RPALETTE, B-209
RSPPPOS, B-215, B-235
RSprite, B-216
RUN, B-4, B-52, B-59, B-65,
B-69, B-71, B-110, B-111,
B-128, B-135, B-158,
B-184, B-194, B-197,
B-205, B-209, B-210,
B-217, B-232, B-244,
B-249, B-252, B-253
RWINDOW, B-218
SAVE, 6-8
SAVEIFF, B-220
SCREEN, B-44, B-50, B-64,
B-82, B-95, B-119, B-126,
B-144, B-150, B-156,
B-174, B-176, B-177,
B-179, B-186, B-205,
B-209, B-210, B-220,
B-221, B-225, B-262
SETBIT, B-227
SOUND, B-231
SPEED, B-233
SPRCOLOR, B-234
STR, B-241
TEMPO, B-99, B-183, B-246,
B-263
TRAP, B-92, B-100, B-102,
B-203, B-251
TRON, B-252, B-253
UNLOCK, B-255
USR, B-259
VOL, B-99, B-183, B-246, B-263
WAIT, B-265
WINDOW, B-267
WPOKE, B-269
XOR, B-270
BASIC 65 Functions
ABS, B-18
ASC, B-21
ATN, B-22
CHR\$, B-47
CLRBIT, B-53
COS, B-62
DEC, B-68
ERR\$, B-102
EXP, B-104
FN, B-69, B-110
FRE, B-115
FREAD, B-116
FWRITE, B-118
HEX\$, B-129
INT, B-137
JOY, B-138
LEFT\$, B-141
LEN, B-142
LOG, B-152
LOG10, B-153
LPEN, B-155
MEM, B-156
MID\$, B-158
MOD, B-160
PEEK, B-178
PIXEL, B-180
POINTER, B-184
POKE, B-185
POS, B-187

POT, B-188	BBR0, H-67
RCOLOR, B-193	BBR1, H-68
RGRAPHIC, B-205	BBR2, H-68
RIGHT\$, B-206	BBR3, H-69
RND, 8-45, B-208	BBR4, H-69
RPALETTE, B-209	BBR5, H-69
RPEN, B-210	BBR6, H-70
RPLAY, B-211	BBR7, H-70
RSPCOLOR, B-213	BBS0, H-71
RSPEED, B-214	BBS1, H-71
RSPPOS, B-215	BBS2, H-72
RSprite, B-216	BBS3, H-72
RWINDOW, B-218	BBS4, H-73
SETBIT, B-227	BBS5, H-73
SGN, B-228	BBS6, H-74
SIN, B-229	BBS7, H-74
SPC, B-232	BCC, H-21, H-75
SQR, B-237	BCS, H-22, H-75
STR\$, B-241	BEQ, H-22, H-76
TAB, B-244	bgcolor, 17-7
TAN, B-245	BIT, H-23, H-76
USR, B-259	BITMAPS, K-8
VAL, B-260	BITQ, G-14, H-130
WPEEK, B-268	blink, 17-8
WPOKE, B-269	blocked, 8-21
BASIC 65 Operators, B-9	BMI, H-23, H-77
AND, 8-44, B-19	BNE, H-24, H-77
NOT, B-168	BORDER, 8-44
OR, B-173	bordercolor, 17-7
XOR, B-270	box, 17-11
BASIC 65 System Commands	BPL, H-24, H-78
EDIT, B-89	BRA, H-78
BASIC 65 System Variables	BREAKPOINT, K-19
DS, B-84	BRK, H-24, H-79
DS\$, B-85	BSR, H-80
DT\$, B-87	BVC, H-25, H-80
EL, B-92	BVS, H-25, H-80
ER, B-100	cellcolor, 17-9
ST, B-238	cgetc, 17-19
TI, B-248	character, 8-13
TI\$, B-249	character set, 8-13
BASIC 65 Variables, B-7	cinput, 17-20

CLC, H-26, H-81
CLD, H-26, H-81
CLE, H-82
clearattr, 17-9
CLI, H-27, H-82
clrscr, 17-5
CLV, H-27, H-83
CMP, H-27, H-50, H-83, H-131
CMPQ, H-131
COMPARE, K-9
conionit, 17-3
Connections
 IEC, 2-7, 6-3
CONT, 8-39
context dependent, 8-19
copyright, ii
CPQ, G-14
cprintf, 17-18
CPUHISTORY, K-23
CPUMEMORY, K-18
cputc, 17-16
cputcxy, 17-15, 17-17
cputdec, 17-17
cputhex, 17-16
cputnc, 17-16
cputncxy, 17-18
cputs, 17-17
cputsxy, 17-17
CPX, H-28, H-84
CPY, H-29, H-85
CPZ, H-85
cross-bar switch, audio, R-23

DCP, H-29
DEBUGCPU, Q-11
DEBUGMON, K-21
DEBUGVIC, Q-11
DEC, H-30, H-86
DEQ, G-14, H-132
DEW, H-87
DEX, H-31, H-88
DEY, H-31, H-88
DEZ, H-89

Digital Audio, L-20
digital video, M-9
Direct Mode, 8-18
DISASSEMBLE, K-9, K-19
Disk Drives, 6-3
 Assignment, 6-5
 Connecting, 2-4
 D81 Images, 4-16, 6-6
 Terminology, 6-3
Display
 Connecting, 2-4
 Setting PAL/NTSC, 4-10, 4-17
DMA
 Inline DMA Lists, L-19
DMA Audio, L-20

END, 8-39
EOM, H-89
EOR, H-32, H-90, H-133
EORQ, G-14, H-134
Errors
 Extra Ignored, 8-22
 Illegal Direct, 8-18
 Syntax, 8-4
 Type mismatch, 8-17
EXIT, K-14
Extra Ignored, 8-22

FILL, K-10, K-20
fillrect, 17-11
FLAGWATCH, K-20
Flash Menu, I-5
flushkeybuf, 17-20
FOR, 8-7
FOREGROUND, 8-44
FRAME1K, Q-11
FRAME2K, Q-11
Freeze Menu, I-5

Games
 Guess the number, 8-33
getcharsetaddr, 17-5
getcolramoffset, 17-4

getkeymodstate, 17-20
getmapedpal, 17-10
getpalbank, 17-10
getpalbanka, 17-10
getscreenaddr, 17-4
getscreensize, 17-5
GO, K-11
gohome, 17-12
GOTO, 8-46
gotox, 17-13
gotoxy, 17-12
gotoy, 17-13
Guess the number, 8-33

HELP, K-19
highlight, 17-8
hline, 17-12
Hot Registers, M-18
HUNT, K-11
HYPERTRAP, K-17
Hippo Error Codes, J-5, J-7
 \$01, J-7
 \$02, J-7
 \$03, J-7
 \$04, J-7
 \$05, J-7
 \$06, J-7
 \$07, J-7
 \$08, J-7, J-32
 \$10, J-7, J-10, J-32, J-38, J-45,
 J-47, J-55
 \$11, J-8
 \$20, J-8
 \$21, J-8
 \$80, J-8, J-37
 \$81, J-8, J-38
 \$82, J-8
 \$83, J-8
 \$84, J-8, J-27, J-28, J-30, J-31
 \$85, J-8, J-33
 \$86, J-8, J-31
 \$87, J-8, J-11, J-30

 \$88, J-8, J-18-J-20, J-27, J-28,
 J-40
 \$89, J-8, J-34
 \$8A, J-8
 \$8B, J-8
 \$8C, J-8
 \$8D, J-8, J-29, J-36
 \$8E, J-8
 \$FF, J-8

Hippo Move to Root Directory, J-12
Hippo Services, J-3
 \$D640 \$00, J-9
 \$D640 \$02, J-25
 \$D640 \$04, J-23
 \$D640 \$06, J-37
 \$D640 \$08, J-26
 \$D640 \$0A, J-24
 \$D640 \$0C, J-11
 \$D640 \$0E, J-29
 \$D640 \$10, J-36
 \$D640 \$12, J-30
 \$D640 \$14, J-32
 \$D640 \$16, J-15
 \$D640 \$18, J-31
 \$D640 \$1A, J-34
 \$D640 \$1C, J-39
 \$D640 \$1E, J-29
 \$D640 \$20, J-16
 \$D640 \$22, J-14
 \$D640 \$24, J-36
 \$D640 \$26, J-36
 \$D640 \$28, J-22
 \$D640 \$2A, J-36
 \$D640 \$2C, J-17
 \$D640 \$2E, J-38
 \$D640 \$30, J-19
 \$D640 \$32, J-20
 \$D640 \$34, J-18
 \$D640 \$36, J-27
 \$D640 \$3A, J-10
 \$D640 \$3E, J-28
 \$D640 \$40, J-40

\$D640 \$42, J-42
\$D640 \$44, J-43
\$D640 \$46, J-41
\$D640 \$48, J-46
\$D640 \$50, J-48
\$D640 \$52, J-52
\$D640 \$54, J-48
\$D640 \$56, J-59
\$D640 \$58, J-48
\$D640 \$60, J-56
\$D640 \$62, J-44
\$D640 \$64, J-48
\$D640 \$66, J-44
\$D640 \$68, J-44
\$D640 \$6A, J-44
\$D640 \$6C, J-56
\$D640 \$6E, J-44
\$D640 \$70, J-58
\$D640 \$72, J-57
\$D640 \$74, J-45
\$D640 \$76, J-55
\$D640 \$7C, J-54
\$D640 \$7E, J-49
\$D641 \$00, J-51
\$D641 \$02, J-50
\$D642 \$00, J-60
\$D642 \$02, J-60
\$D642 \$04, J-60
\$D642 \$06, J-60
\$D642 \$10, J-61
\$D642 \$12, J-61
\$D642 \$14, J-61
\$D642 \$16, J-61
\$D643 \$38, J-7
\$D643 \$3E, J-12
\$D643 \$xx, J-53
\$D67F \$xx, J-61

I/O
 blocking, 8-21

IF, 8-30

Illegal Direct Error, 8-18

IMDH™, M-9

INC, H-32, H-91
Inline DMA Lists, L-19
INPUT, 8-17, 8-24
INQ, G-14, H-134
Integrated Marvellous Digital
 Hookup™, M-9

INTERRUPTS, K-21

INW, H-91
INX, H-33, H-92
INY, H-33, H-92
INZ, H-93
ISC, H-34

JMP, H-35, H-93
JSR, H-35, H-94
JUMP, K-11

kbhit, 17-19

Keyboard
 ALT, 3-7
 Arrow Keys, 3-5
 CAPS LOCK, 3-7
 CLR HOME, 3-5
 CTRL, 3-4, 3-8, C-5
 Cursor Keys, 3-5, C-10
 Escape Sequences, 3-10, C-9
 Function Keys, 3-6
 HELP, 3-6
 INST DEL, 3-5
 matrix, D-15
 MEGA Key, 3-6, 3-8, 10-4
 NO SCROLL, 3-6
 PETSCII Codes and CHR\$, B-47,
 C-3
 RESTORE, 3-4
 RETURN, 3-3
 RUN STOP, 3-4
 Shift Keys, 3-3, 3-6, C-8
 SHIFT LOCK, 3-3

KIL, H-36

LAS, H-37
LAX, H-37

LDA, H-38, H-95, H-135
LDQ, G-14, H-136
LDX, H-39, H-95
LDY, H-40, H-96
LDZ, H-97
LET, 8-15
light pen, R-21
Line Drawing, L-15
 DMA Option Bytes, L-15
Lines
 editing, 8-25
 renumbering, 8-38
 replacing, 8-25
LIST, 8-11, 8-12
LOAD, K-11
LOADMEMORY, K-21
LSR, H-40, H-97
LSRQ, G-14, H-136

MAP, G-8-G-10, H-98
MEGA Flash, I-5
MEMORY, K-12, K-21
Memory banking, G-8-G-10
mixer, audio, R-23
MOD-file style audio, L-20
MONITOR
 Matrix Mode/Serial Monitor, K-15
 MEGA65 Monitor, K-3
MONITOR Commands, K-15
 ., K-14
 >, K-14
 ASSEMBLE, K-6
 BITMAPS, K-8
 BREAKPOINT, K-19
 COMPARE, K-9
 CPUHISTORY, K-23
 CPUMEMORY, K-18
 DEBUGMON, K-21
 DISASSEMBLE, K-9, K-19
 EXIT, K-14
 FILL, K-10, K-20
 FLAGWATCH, K-20
 GO, K-11
HELP, K-19
HUNT, K-11
HYPERTRAP, K-17
INTERRUPTS, K-21
JUMP, K-11
LOAD, K-11
LOADMEMORY, K-21
MEMORY, K-12, K-21
REGISTERS, K-13, K-22
SAVE, K-13
SETMEMORY, K-22
SETPC, K-20
TRACE, K-22
TRANSFER, K-13
UARTDIVISOR, K-18
VERIFY, K-13
WATCHPOINT, K-22
movedown, 17-14
moveleft, 17-14
moveright, 17-14
moveup, 17-13

name spaces, 8-17
NEG, H-99
NO SCROLL, 8-47
NOP, H-41
not equal, 8-31

OpenROMs, 13-4
operators
 relational, 8-30
ORA, H-43, H-99, H-137
ORQ, G-14, H-138

pcprintf, 17-19
pcputc, 17-15
pcputs, 17-16
pcputsxy, 17-15
PHA, H-43, H-100
PHP, H-44, H-100
PHW, H-101
PHX, H-101
PHY, H-102

PHZ, H-102	\$D017, M-37
PLA, H-44, H-102	\$D018, M-37
PLP, H-44, H-103	\$D019, M-37
PLX, H-103	\$D01A, M-37
PLY, H-103	\$D01B, M-37
PLZ, H-104	\$D01C, M-37
PORT, G-16	\$D01D, M-37
PORTDDR, G-16	\$D01E, M-37
PRINT, 8-11	\$D01F, M-37
Programmes	\$D020, M-37, M-40, M-42
editing, 8-25	\$D021, M-37, M-40, M-42
replacing lines, 8-25	\$D022, M-37, M-40, M-42
quote mode, 8-35	\$D023, M-37, M-40, M-42
Real-Time Clock	\$D024, M-38, M-40, M-42
Replacing the Battery, 2-7	\$D025, M-38, M-40, M-42
Setting the Date/Time, 4-16	\$D026, M-38, M-40, M-42
REGISTERS, K-13, K-22	\$D027, M-38
Registers	\$D028, M-38
\$D000, M-37	\$D029, M-38
\$D001, M-37	\$D02A, M-38
\$D002, M-37	\$D02B, M-38
\$D003, M-37	\$D02C, M-38
\$D004, M-37	\$D02D, M-38
\$D005, M-37	\$D02E, M-38
\$D006, M-37	\$D02F, M-40, M-42
\$D007, M-37	\$D030, M-38, M-40
\$D008, M-37	\$D031, M-40
\$D009, M-37	\$D033, M-40
\$D00A, M-37	\$D034, M-40
\$D00B, M-37	\$D035, M-40
\$D00C, M-37	\$D036, M-40
\$D00D, M-37	\$D037, M-40
\$D00E, M-37	\$D038, M-40
\$D00F, M-37	\$D039, M-40
\$D010, M-37	\$D03A, M-40
\$D011, M-37	\$D03B, M-40
\$D012, M-37	\$D03C, M-40
\$D013, M-37	\$D03D, M-40
\$D014, M-37	\$D03E, M-40
\$D015, M-37	\$D03F, M-40
\$D016, M-37	\$D040, M-40
	\$D041, M-40

\$D042, M-40	\$D06E, M-43
\$D043, M-40	\$D06F, M-43
\$D044, M-40	\$D070, M-43
\$D045, M-40	\$D071, M-43
\$D046, M-40	\$D072, M-43
\$D047, M-40	\$D073, M-43
\$D048, M-42	\$D074, M-43
\$D049, M-42	\$D075, M-43
\$D04A, M-42	\$D076, M-43
\$D04B, M-42	\$D077, M-43
\$D04C, M-42	\$D078, M-43
\$D04D, M-42	\$D079, M-43
\$D04E, M-42	\$D07A, M-43
\$D04F, M-42	\$D07B, M-43
\$D050, M-42	\$D07C, M-43
\$D051, M-42	\$D080, R-11
\$D052, M-42	\$D081, R-11
\$D053, M-42	\$D082, R-11
\$D054, M-42	\$D083, R-11
\$D055, M-42	\$D084, R-11
\$D056, M-42	\$D085, R-11
\$D057, M-42	\$D086, R-11
\$D058, M-43	\$D087, R-11
\$D059, M-43	\$D088, R-11
\$D05A, M-43	\$D089, R-11
\$D05B, M-43	\$D08A, R-11
\$D05C, M-43	\$D100 - \$D1FF, M-40
\$D05D, M-43	\$D200 - \$D2FF, M-40
\$D05E, M-43	\$D300 - \$D3FF, M-40
\$D05F, M-43	\$D400, N-3
\$D060, M-43	\$D401, N-3
\$D061, M-43	\$D402, N-3
\$D062, M-43	\$D403, N-3
\$D063, M-43	\$D404, N-3
\$D064, M-43	\$D405, N-3
\$D065, M-43	\$D406, N-3
\$D068, M-43	\$D407, N-3
\$D069, M-43	\$D408, N-3
\$D06A, M-43	\$D409, N-3
\$D06B, M-43	\$D40A, N-3
\$D06C, M-43	\$D40B, N-3
\$D06D, M-43	\$D40C, N-3

\$D40D, N-3	\$D622, P-4
\$D40E, N-3	\$D623, P-4
\$D40F, N-3	\$D625, P-4
\$D410, N-3	\$D626, P-4
\$D411, N-3	\$D627, P-4
\$D412, N-3	\$D628, P-4
\$D413, N-3	\$D629, P-5
\$D414, N-3	\$D63C, N-4
\$D415, N-3	\$D640, G-17, G-29, J-3
\$D416, N-3	\$D641, G-17, G-29, J-3
\$D417, N-3	\$D642, G-17, J-3
\$D418, N-3	\$D643, G-17, G-29, J-3
\$D419, N-3	\$D644, G-17, G-29, J-3
\$D41A, N-3	\$D645, G-17, G-29, J-3
\$D41B, N-3	\$D646, G-17, G-29, J-3
\$D41C, N-4	\$D647, G-17, G-29, J-3
\$D600, P-3	\$D648, G-17, G-29, J-3
\$D601, P-3	\$D649, G-17, G-29, J-3
\$D602, P-3	\$D64A, G-17, G-29, J-3
\$D603, P-3	\$D64B, G-17, G-29, J-3
\$D604, P-3	\$D64C, G-17, G-29, J-3
\$D605, P-3	\$D64D, G-17, G-29, J-3
\$D606, P-3	\$D64E, G-17, G-29, J-3
\$D609, P-4	\$D64F, G-17, G-29, J-3
\$D60B, P-4	\$D650, G-17, G-29, J-3
\$D60C, P-4	\$D651, G-17, G-29, J-3
\$D60D, P-4	\$D652, G-17, G-29, J-3
\$D60E, P-4	\$D653, G-17, G-29, J-3
\$D60F, P-4	\$D654, G-17, G-29, J-3
\$D610, P-4	\$D655, G-17, G-29, J-3
\$D611, P-4	\$D656, G-17, G-30, J-3
\$D612, P-4	\$D657, G-17, G-30, J-3
\$D615, P-4	\$D658, G-17, G-30, J-3
\$D616, P-4	\$D659, G-17, G-30, J-3
\$D617, P-4	\$D65A, G-17, J-3
\$D618, P-4	\$D65B, G-17, J-3
\$D619, P-4	\$D65C, G-17, J-3
\$D61A, P-4	\$D65D, G-17, J-3
\$D61D, P-4	\$D65E, G-17, J-3
\$D61E, P-4	\$D65F, G-17, J-3
\$D620, P-4	\$D660, G-17, J-3
\$D621, P-4	\$D661, G-17, J-3

\$D662, G-18, J-3	\$D690, R-15
\$D663, G-18, J-3	\$D691, R-15
\$D664, G-18, J-3	\$D692, R-15
\$D665, G-18, J-3	\$D693, R-15
\$D666, G-18, J-3	\$D6A0, R-13
\$D667, G-18, J-3	\$D6A1, R-15
\$D668, G-18, J-3	\$D6A2, R-13
\$D669, G-18, J-3	\$D6AE, R-19
\$D66A, G-18, J-3	\$D6AF, R-19
\$D66B, G-18, J-3	\$D6B0, R-21
\$D66C, G-18, J-3	\$D6B1, R-22
\$D66D, G-18, J-3	\$D6B2, R-22
\$D66E, G-18, J-3	\$D6B3, R-22
\$D66F, G-18, J-3	\$D6B4, R-22
\$D670, G-18, G-30, J-3	\$D6B5, R-22
\$D671, G-18, G-30, J-3	\$D6B7, R-22
\$D672, G-18, G-30, J-3	\$D6B8, R-22
\$D673, G-18, J-3	\$D6B9, R-22
\$D674, G-18, J-3	\$D6BA, R-22
\$D675, G-18, J-3	\$D6BB, R-22
\$D676, G-18, J-3	\$D6BC, R-22
\$D677, G-18, J-3	\$D6BD, R-22
\$D678, G-18, J-3	\$D6BE, R-22
\$D679, G-18, J-3	\$D6C0, R-22
\$D67A, G-18, J-3	\$D6E0, Q-9
\$D67B, G-18, J-3	\$D6E1, Q-9
\$D67C, G-18, G-30, J-3	\$D6E2, Q-9
\$D67D, G-18, G-30, J-3	\$D6E3, Q-9
\$D67E, G-18, G-30, J-3	\$D6E4, Q-9
\$D67F, G-18, G-30, J-3	\$D6E5, Q-9
\$D680, R-19	\$D6E6, Q-9
\$D681, R-19	\$D6E7, Q-9
\$D682, R-19	\$D6E8, Q-9
\$D683, R-19	\$D6E9, Q-9
\$D684, R-19	\$D6EA, Q-9
\$D686, R-19	\$D6EB, Q-9
\$D68A, R-14, R-19	\$D6EC, Q-9
\$D68B, R-14	\$D6ED, Q-9
\$D68C, R-14	\$D6EE, Q-9
\$D68D, R-14	\$D6F4, R-25
\$D68E, R-14	\$D6F5, R-25
\$D68F, R-14	\$D6F8, R-25

\$D6F9, R-25	\$D736, L-24
\$D6FA, R-25	\$D737, L-24
\$D6FB, R-25	\$D738, L-24
\$D6FC, R-25	\$D739, L-24
\$D6FD, R-25	\$D73A, L-24
\$D700, L-23	\$D73B, L-24
\$D701, L-23	\$D73C, L-24
\$D702, L-23	\$D73D, L-24
\$D703, L-23	\$D73E, L-24
\$D704, L-23	\$D73F, L-24
\$D705, L-23	\$D740, L-24
\$D706, L-23	\$D741, L-24
\$D70E, L-23	\$D742, L-24
\$D70F, G-20	\$D743, L-24
\$D710, G-18	\$D744, L-24
\$D711, L-23, R-25	\$D745, L-24
\$D71C, L-23	\$D746, L-24
\$D71D, L-23	\$D747, L-24
\$D71E, L-23	\$D748, L-24
\$D71F, L-23	\$D749, L-24
\$D720, L-23	\$D74A, L-24
\$D721, L-23	\$D74B, L-24
\$D722, L-23	\$D74C, L-24
\$D723, L-23	\$D74D, L-25
\$D724, L-23	\$D74E, L-25
\$D725, L-23	\$D74F, L-25
\$D726, L-23	\$D750, L-25
\$D727, L-24	\$D751, L-25
\$D728, L-24	\$D752, L-25
\$D729, L-24	\$D753, L-25
\$D72A, L-24	\$D754, L-25
\$D72B, L-24	\$D755, L-25
\$D72C, L-24	\$D756, L-25
\$D72D, L-24	\$D757, L-25
\$D72E, L-24	\$D758, L-25
\$D72F, L-24	\$D759, L-25
\$D730, L-24	\$D75A, L-25
\$D731, L-24	\$D75B, L-25
\$D732, L-24	\$D75C, L-25
\$D733, L-24	\$D75D, L-25
\$D734, L-24	\$D75E, L-25
\$D735, L-24	\$D75F, L-25

\$D768, G-20	\$D792, G-21
\$D769, G-20	\$D793, G-21
\$D76A, G-20	\$D794, G-21
\$D76B, G-20	\$D795, G-21
\$D76C, G-20	\$D796, G-21
\$D76D, G-20	\$D797, G-21
\$D76E, G-20	\$D798, G-21
\$D76F, G-20	\$D799, G-21
\$D770, G-20	\$D79A, G-21
\$D771, G-20	\$D79B, G-21
\$D772, G-20	\$D79C, G-21
\$D773, G-20	\$D79D, G-21
\$D774, G-20	\$D79E, G-21
\$D775, G-20	\$D79F, G-21
\$D776, G-20	\$D7A0, G-21
\$D777, G-20	\$D7A1, G-21
\$D778, G-20	\$D7A2, G-21
\$D779, G-20	\$D7A3, G-21
\$D77A, G-20	\$D7A4, G-21
\$D77B, G-20	\$D7A5, G-21
\$D77C, G-20	\$D7A6, G-21
\$D77D, G-20	\$D7A7, G-21
\$D77E, G-20	\$D7A8, G-21
\$D77F, G-20	\$D7A9, G-21
\$D780, G-20	\$D7AA, G-21
\$D781, G-20	\$D7AB, G-21
\$D782, G-20	\$D7AC, G-21
\$D783, G-20	\$D7AD, G-21
\$D784, G-20	\$D7AE, G-21
\$D785, G-20	\$D7AF, G-21
\$D786, G-20	\$D7B0, G-22
\$D787, G-20	\$D7B1, G-22
\$D788, G-20	\$D7B2, G-22
\$D789, G-20	\$D7B3, G-22
\$D78A, G-21	\$D7B4, G-22
\$D78B, G-21	\$D7B5, G-22
\$D78C, G-21	\$D7B6, G-22
\$D78D, G-21	\$D7B7, G-22
\$D78E, G-21	\$D7B8, G-22
\$D78F, G-21	\$D7B9, G-22
\$D790, G-21	\$D7BA, G-22
\$D791, G-21	\$D7BB, G-22

\$D7BC, G-22	\$D7FB, G-18
\$D7BD, G-22	\$D7FD, G-18
\$D7BE, G-22	\$D7FE, G-18
\$D7BF, G-22	\$DC00, O-3
\$D7C0, G-22	\$DC01, O-3
\$D7C1, G-22	\$DC02, O-3
\$D7C2, G-22	\$DC03, O-3
\$D7C3, G-22	\$DC04, O-3
\$D7C4, G-22	\$DC05, O-3
\$D7C5, G-22	\$DC06, O-3
\$D7C6, G-22	\$DC07, O-3
\$D7C7, G-22	\$DC08, O-3
\$D7C8, G-22	\$DC09, O-3
\$D7C9, G-22	\$DC0A, O-3
\$D7CA, G-22	\$DC0B, O-3
\$D7CB, G-22	\$DC0C, O-3
\$D7CC, G-22	\$DC0D, O-3
\$D7CD, G-22	\$DC0E, O-3
\$D7CE, G-22	\$DC0F, O-3
\$D7CF, G-22	\$DC10, O-6
\$D7D0, G-22	\$DC11, O-6
\$D7D1, G-22	\$DC12, O-6
\$D7D2, G-22	\$DC13, O-7
\$D7D3, G-23	\$DC14, O-7
\$D7D4, G-23	\$DC15, O-7
\$D7D5, G-23	\$DC16, O-7
\$D7D6, G-23	\$DC17, O-7
\$D7D7, G-23	\$DC18, O-7
\$D7D8, G-23	\$DC19, O-7
\$D7D9, G-23	\$DC1A, O-7
\$D7DA, G-23	\$DC1B, O-7
\$D7DB, G-23	\$DC1C, O-7
\$D7DC, G-23	\$DC1D, O-7
\$D7DD, G-23	\$DC1E, O-7
\$D7DE, G-23	\$DC1F, O-7
\$D7DF, G-23	\$DD00, O-4
\$D7E0, G-23	\$DD01, O-4
\$D7E1, G-23	\$DD02, O-4
\$D7E2, G-23	\$DD03, O-5
\$D7E3, G-23	\$DD04, O-5
\$D7EF, G-18	\$DD05, O-5
\$D7FA, G-18	\$DD06, O-5

\$DD07, O-5	53263, M-37
\$DD08, O-5	53264, M-37
\$DD09, O-5	53265, M-37
\$DD0B, O-5	53266, M-37
\$DD0C, O-5	53267, M-37
\$DD0D, O-5	53268, M-37
\$DD0E, O-5	53269, M-37
\$DD0F, O-5	53270, M-37
\$DD10, O-8	53271, M-37
\$DD11, O-8	53272, M-37
\$DD12, O-8	53273, M-37
\$DD13, O-8	53274, M-37
\$DD14, O-8	53275, M-37
\$DD15, O-8	53276, M-37
\$DD16, O-8	53277, M-37
\$DD17, O-8	53278, M-37
\$DD18, O-8	53279, M-37
\$DD19, O-8	53280, M-37, M-40, M-42
\$DD1A, O-8	53281, M-37, M-40, M-42
\$DD1B, O-8	53282, M-37, M-40, M-42
\$DD1C, O-8	53283, M-37, M-40, M-42
\$DD1D, O-8	53284, M-38, M-40, M-42
\$DD1E, O-8	53285, M-38, M-40, M-42
\$DD1F, O-8	53286, M-38, M-40, M-42
53248, M-37	53287, M-38
53249, M-37	53288, M-38
53250, M-37	53289, M-38
53251, M-37	53290, M-38
53252, M-37	53291, M-38
53253, M-37	53292, M-38
53254, M-37	53293, M-38
53255, M-37	53294, M-38
53256, M-37	53295, M-40, M-42
53257, M-37	53296, M-38, M-40
53258, M-37	53297, M-40
53259, M-37	53299, M-40
53260, M-37	53300, M-40
53261, M-37	53301, M-40
53262, M-37	53302, M-40
	53303, M-40
	53304, M-40
	53305, M-40

53306, M-40	53348, M-43
53307, M-40	53349, M-43
53308, M-40	53352, M-43
53309, M-40	53353, M-43
53310, M-40	53354, M-43
53311, M-40	53355, M-43
53312, M-40	53356, M-43
53313, M-40	53357, M-43
53314, M-40	53358, M-43
53315, M-40	53359, M-43
53316, M-40	53360, M-43
53317, M-40	53361, M-43
53318, M-40	53362, M-43
53319, M-40	53363, M-43
53320, M-42	53364, M-43
53321, M-42	53365, M-43
53322, M-42	53366, M-43
53323, M-42	53367, M-43
53324, M-42	53368, M-43
53325, M-42	53369, M-43
53326, M-42	53370, M-43
53327, M-42	53371, M-43
53328, M-42	53372, M-43
53329, M-42	53376, R-11
53330, M-42	53377, R-11
53331, M-42	53378, R-11
53332, M-42	53379, R-11
53333, M-42	53380, R-11
53334, M-42	53381, R-11
53335, M-42	53382, R-11
53336, M-43	53383, R-11
53337, M-43	53384, R-11
53338, M-43	53385, R-11
53339, M-43	53386, R-11
53340, M-43	54272, N-3
53341, M-43	54273, N-3
53342, M-43	54274, N-3
53343, M-43	54275, N-3
53344, M-43	54276, N-3
53345, M-43	54277, N-3
53346, M-43	54278, N-3
53347, M-43	54279, N-3

54280, N-3	54810, P-4
54281, N-3	54813, P-4
54282, N-3	54814, P-4
54283, N-3	54816, P-4
54284, N-3	54817, P-4
54285, N-3	54818, P-4
54286, N-3	54819, P-4
54287, N-3	54821, P-4
54288, N-3	54822, P-4
54289, N-3	54823, P-4
54290, N-3	54824, P-4
54291, N-3	54825, P-5
54292, N-3	54844, N-4
54293, N-3	54848, G-17, G-29
54294, N-3	54849, G-17, G-29
54295, N-3	54850, G-17
54296, N-3	54851, G-17, G-29
54297, N-3	54852, G-17, G-29
54298, N-3	54853, G-17, G-29
54299, N-3	54854, G-17, G-29
54300, N-4	54855, G-17, G-29
54784, P-3	54856, G-17, G-29
54785, P-3	54857, G-17, G-29
54786, P-3	54858, G-17, G-29
54787, P-3	54859, G-17, G-29
54788, P-3	54860, G-17, G-29
54789, P-3	54861, G-17, G-29
54790, P-3	54862, G-17, G-29
54793, P-4	54863, G-17, G-29
54795, P-4	54864, G-17, G-29
54796, P-4	54865, G-17, G-29
54797, P-4	54866, G-17, G-29
54798, P-4	54867, G-17, G-29
54799, P-4	54868, G-17, G-29
54800, P-4	54869, G-17, G-29
54801, P-4	54870, G-17, G-30
54802, P-4	54871, G-17, G-30
54805, P-4	54872, G-17, G-30
54806, P-4	54873, G-17, G-30
54807, P-4	54874, G-17
54808, P-4	54875, G-17
54809, P-4	54876, G-17

54877, G-17	54923, R-14
54878, G-17	54924, R-14
54879, G-17	54925, R-14
54880, G-17	54926, R-14
54881, G-17	54927, R-14
54882, G-18	54928, R-15
54883, G-18	54929, R-15
54884, G-18	54930, R-15
54885, G-18	54931, R-15
54886, G-18	54944, R-13
54887, G-18	54945, R-15
54888, G-18	54946, R-13
54889, G-18	54958, R-19
54890, G-18	54959, R-19
54891, G-18	54960, R-21
54892, G-18	54961, R-22
54893, G-18	54962, R-22
54894, G-18	54963, R-22
54895, G-18	54964, R-22
54896, G-18, G-30	54965, R-22
54897, G-18, G-30	54967, R-22
54898, G-18, G-30	54968, R-22
54899, G-18	54969, R-22
54900, G-18	54970, R-22
54901, G-18	54971, R-22
54902, G-18	54972, R-22
54903, G-18	54973, R-22
54904, G-18	54974, R-22
54905, G-18	54976, R-22
54906, G-18	55008, Q-9
54907, G-18	55009, Q-9
54908, G-18, G-30	55010, Q-9
54909, G-18, G-30	55011, Q-9
54910, G-18, G-30	55012, Q-9
54911, G-18, G-30	55013, Q-9
54912, R-19	55014, Q-9
54913, R-19	55015, Q-9
54914, R-19	55016, Q-9
54915, R-19	55017, Q-9
54916, R-19	55018, Q-9
54918, R-19	55019, Q-9
54922, R-14, R-19	55020, Q-9

55021, Q-9	55089, L-24
55022, Q-9	55090, L-24
55028, R-25	55091, L-24
55029, R-25	55092, L-24
55032, R-25	55093, L-24
55033, R-25	55094, L-24
55034, R-25	55095, L-24
55035, R-25	55096, L-24
55036, R-25	55097, L-24
55037, R-25	55098, L-24
55040, L-23	55099, L-24
55041, L-23	55100, L-24
55042, L-23	55101, L-24
55043, L-23	55102, L-24
55044, L-23	55103, L-24
55045, L-23	55104, L-24
55046, L-23	55105, L-24
55054, L-23	55106, L-24
55055, G-20	55107, L-24
55056, G-18	55108, L-24
55057, L-23, R-25	55109, L-24
55068, L-23	55110, L-24
55069, L-23	55111, L-24
55070, L-23	55112, L-24
55071, L-23	55113, L-24
55072, L-23	55114, L-24
55073, L-23	55115, L-24
55074, L-23	55116, L-24
55075, L-23	55117, L-25
55076, L-23	55118, L-25
55077, L-23	55119, L-25
55078, L-23	55120, L-25
55079, L-24	55121, L-25
55080, L-24	55122, L-25
55081, L-24	55123, L-25
55082, L-24	55124, L-25
55083, L-24	55125, L-25
55084, L-24	55126, L-25
55085, L-24	55127, L-25
55086, L-24	55128, L-25
55087, L-24	55129, L-25
55088, L-24	55130, L-25

55131, L-25	55181, G-21
55132, L-25	55182, G-21
55133, L-25	55183, G-21
55134, L-25	55184, G-21
55135, L-25	55185, G-21
55144, G-20	55186, G-21
55145, G-20	55187, G-21
55146, G-20	55188, G-21
55147, G-20	55189, G-21
55148, G-20	55190, G-21
55149, G-20	55191, G-21
55150, G-20	55192, G-21
55151, G-20	55193, G-21
55152, G-20	55194, G-21
55153, G-20	55195, G-21
55154, G-20	55196, G-21
55155, G-20	55197, G-21
55156, G-20	55198, G-21
55157, G-20	55199, G-21
55158, G-20	55200, G-21
55159, G-20	55201, G-21
55160, G-20	55202, G-21
55161, G-20	55203, G-21
55162, G-20	55204, G-21
55163, G-20	55205, G-21
55164, G-20	55206, G-21
55165, G-20	55207, G-21
55166, G-20	55208, G-21
55167, G-20	55209, G-21
55168, G-20	55210, G-21
55169, G-20	55211, G-21
55170, G-20	55212, G-21
55171, G-20	55213, G-21
55172, G-20	55214, G-21
55173, G-20	55215, G-21
55174, G-20	55216, G-22
55175, G-20	55217, G-22
55176, G-20	55218, G-22
55177, G-20	55219, G-22
55178, G-21	55220, G-22
55179, G-21	55221, G-22
55180, G-21	55222, G-22

55223, G-22	55265, G-23
55224, G-22	55266, G-23
55225, G-22	55267, G-23
55226, G-22	55279, G-18
55227, G-22	55290, G-18
55228, G-22	55291, G-18
55229, G-22	55293, G-18
55230, G-22	55294, G-18
55231, G-22	56320, O-3
55232, G-22	56321, O-3
55233, G-22	56322, O-3
55234, G-22	56323, O-3
55235, G-22	56324, O-3
55236, G-22	56325, O-3
55237, G-22	56326, O-3
55238, G-22	56327, O-3
55239, G-22	56328, O-3
55240, G-22	56329, O-3
55241, G-22	56330, O-3
55242, G-22	56331, O-3
55243, G-22	56332, O-3
55244, G-22	56333, O-3
55245, G-22	56334, O-3
55246, G-22	56335, O-3
55247, G-22	56336, O-6
55248, G-22	56337, O-6
55249, G-22	56338, O-6
55250, G-22	56339, O-7
55251, G-23	56340, O-7
55252, G-23	56341, O-7
55253, G-23	56342, O-7
55254, G-23	56343, O-7
55255, G-23	56344, O-7
55256, G-23	56345, O-7
55257, G-23	56346, O-7
55258, G-23	56347, O-7
55259, G-23	56348, O-7
55260, G-23	56349, O-7
55261, G-23	56350, O-7
55262, G-23	56351, O-7
55263, G-23	56576, O-4
55264, G-23	56577, O-4

56578, O-4
56579, O-5
56580, O-5
56581, O-5
56582, O-5
56583, O-5
56584, O-5
56585, O-5
56587, O-5
56588, O-5
56589, O-5
56590, O-5
56591, O-5
56592, O-8
56593, O-8
56594, O-8
56595, O-8
56596, O-8
56597, O-8
56598, O-8
56599, O-8
56600, O-8
56601, O-8
56602, O-8
56603, O-8
56604, O-8
56605, O-8
56606, O-8
56607, O-8
ABTPALSEL, M-43
ACCESSKEY, P-5
ADDRBANK, L-23
ADDRLSB, L-23, L-25
ADDRLSBTRIG, L-23
ADDRMB, L-23, L-25
ADDRMSB, L-23
ALGO, R-11
ALPHADELAY, M-43, M-44
ALPHEN, M-44
ALRM, O-3, O-5
ALRMAMPM, O-7, O-8
ALRMHOUR, O-7, O-8
ALRMJIF, O-7, O-8
ALRMMIN, O-7, O-8
ALRMSEC, O-7, O-8
ALT, R-11
ASCFAST, G-30
ASCIIKEY, P-4, P-5
ATTR, M-40
AUDBLKTO, L-23, L-25
AUDEN, L-25
AUDWRBLK, L-25
AUTO2XSEL, R-19
B1ADEVN, M-40
B1ADODD, M-40
B1PIX, M-40
B2ADEVN, M-40
B2ADODD, M-40
B2PIX, M-40
B3ADEVN, M-40
B3ADODD, M-40
B3PIX, M-40
B4ADEVN, M-40
B4ADODD, M-40
B4PIX, M-40
B5ADEVN, M-40
B5ADODD, M-40
B5PIX, M-40
B6ADEVN, M-40
B6ADODD, M-40
B6PIX, M-40
B7ADEVN, M-40
B7ADODD, M-40
B7PIX, M-40
BADEXTRA, G-18, G-19
BADLEN, G-19
BASHDDR, P-4, P-5
BBDRPOS, M-42, M-44
BCST, Q-9
BITPBANK, M-43, M-44
BLKD, L-25
BLNK, M-38
BMM, M-38
BNPIX, M-40, M-41

BORDERCOL, [M-37](#), [M-38](#),
[M-40](#)-[M-42](#), [M-44](#)
BP16ENS, [M-43](#), [M-44](#)
BPCOMP, [M-40](#), [M-41](#)
BPM, [M-41](#)
BPX, [M-40](#), [M-41](#)
BPY, [M-40](#), [M-41](#)
BRCOST, [G-19](#)
BSP, [M-37](#), [M-38](#)
BTPALSEL, [M-43](#), [M-44](#)
BUSY, [R-11](#)
BXADEVN, [M-40](#), [M-41](#)
BXADODD, [M-40](#), [M-41](#)
C128FAST, [M-38](#)
CALCEN, [G-23](#)
CALXDELTALSB, [R-22](#)
CALXSCALELSB, [R-22](#)
CALXSCALEMSB, [R-22](#)
CALYDELTALSB, [R-22](#)
CALYDELTAMSB, [R-22](#)
CALYSCALELSB, [R-22](#)
CALYSCALEMSB, [R-22](#)
CARTEN, [G-19](#)
CB, [M-37](#), [M-38](#)
CDC00, [R-19](#)
CH0RVOL, [L-23](#), [L-25](#)
CH1BADDRC, [L-24](#)
CH1BADDRL, [L-24](#)
CH1BADDRM, [L-24](#)
CH1CURADDRC, [L-24](#)
CH1CURADDRL, [L-24](#)
CH1CURADDRM, [L-24](#)
CH1FREQC, [L-24](#)
CH1FREQL, [L-24](#)
CH1FREQM, [L-24](#)
CH1RVOL, [L-23](#), [L-25](#)
CH1SBITS, [L-24](#)
CH1TADDRL, [L-24](#)
CH1TADDRM, [L-24](#)
CH1TMRADDRC, [L-24](#)
CH1TMRADDRL, [L-24](#)
CH1TMRADDRM, [L-24](#)
CH1VOLUME, [L-24](#)
CH2BADDRC, [L-24](#)
CH2BADDRL, [L-24](#)
CH2BADDRM, [L-24](#)
CH2CURADDRC, [L-24](#)
CH2CURADDRL, [L-24](#)
CH2CURADDRM, [L-24](#)
CH2FREQC, [L-24](#)
CH2FREQL, [L-24](#)
CH2FREQM, [L-24](#)
CH2LVOL, [L-23](#), [L-25](#)
CH2SBITS, [L-24](#)
CH2TADDRL, [L-24](#)
CH2TADDRM, [L-24](#)
CH2TMRADDRC, [L-25](#)
CH2TMRADDRL, [L-25](#)
CH2TMRADDRM, [L-25](#)
CH2VOLUME, [L-24](#)
CH3BADDRC, [L-25](#)
CH3BADDRL, [L-25](#)
CH3BADDRM, [L-25](#)
CH3CURADDRC, [L-25](#)
CH3CURADDRL, [L-25](#)
CH3CURADDRM, [L-25](#)
CH3FREQC, [L-25](#)
CH3FREQL, [L-25](#)
CH3FREQM, [L-25](#)
CH3LVOL, [L-23](#), [L-25](#)
CH3SBITS, [L-25](#)
CH3TADDRL, [L-25](#)
CH3TADDRM, [L-25](#)
CH3TMRADDRC, [L-25](#)
CH3TMRADDRL, [L-25](#)
CH3TMRADDRM, [L-25](#)
CH3VOLUME, [L-25](#)
CHARPTRBNK, [M-43](#), [M-44](#)
CHARPTRLSB, [M-43](#), [M-44](#)
CHARPTRMSB, [M-43](#), [M-44](#)
CHARSZ, [P-3](#)
CHR16, [M-44](#)
CHRCOUNT, [M-43](#), [M-44](#)
CHRXSCL, [M-43](#), [M-44](#)

CHRYSCL, M-43, M-44
CHXBADDRC, L-23, L-25
CHXBADDRL, L-23, L-25
CHXBADDRM, L-23, L-26
CHXCURADDRC, L-24, L-26
CHXCURADDRL, L-24, L-26
CHXCURADDRM, L-24, L-26
CHXEN, L-26
CHXFREQC, L-23, L-26
CHXFREQL, L-23, L-26
CHXFREQM, L-23, L-26
CHXLOOP, L-26
CHXSBITS, L-23, L-26
CHXSGN, L-26
CHXSINE, L-26
CHXSTP, L-26
CHXTADDRL, L-24, L-26
CHXTADDRL, L-24, L-26
CHXTMRADDRC, L-24, L-26
CHXTMRADDRL, L-24, L-26
CHXTMRADDRM, L-24, L-26
CHXVOLUME, L-24, L-26
CLOCK, R-11
CMDANDSTAT, R-19, R-20
COLPTRLSB, M-43, M-44
COLPTRMSB, M-43, M-44
COMMAND, Q-9, R-11
CONN41, P-5
CPUFAST, G-30
CRAM2K, M-41
CRC, R-11
CROM9, M-41
CSEL, M-38
D0D64, R-15
D0IMG, R-15
D0MD, R-15
D0P, R-15
D0STARTSEC0, R-14, R-15
D0STARTSEC1, R-14, R-15
D0STARTSEC2, R-14, R-15
D0STARTSEC3, R-14, R-15
D0WP, R-15
D1D64, R-15
D1IMG, R-15
D1MD, R-15
D1P, R-15
D1STARTSEC0, R-15
D1STARTSEC1, R-15
D1STARTSEC2, R-15
D1STARTSEC3, R-15
D1WP, R-15
DATA, P-3, R-11
DATARATE, R-13
DBGDIR, R-13
DBGMOTORA, R-13
DBGWDATA, R-13
DBGWGATE, R-13
DBLRR, M-44
DD00DELAY, O-7, O-8
DDRA, O-3-O-5
DDR2, O-3, O-5
DEBUGC, M-43, M-44
DENSITY, R-13
DIGILEFTLSB, R-25
DIGILEFTMSB, R-25
DIGILLSB, R-25
DIGILMSB, R-25
DIGIRIGHTLSB, R-25
DIGIRIGHTMSB, R-25
DIGIRLSB, R-25
DIGIRMSB, R-25
DIR, R-11
DISKIN, R-11
DISPROWS, M-43, M-44
DIVBUSY, G-23
DIVISOR, P-3
DIVOUT, G-20, G-23
DMADSTMB, G-29, G-30
DMAADDR, G-29, G-30
DMASRCMB, G-29, G-30
DRQ, R-12
DRXD, Q-9
DRXDV, Q-9
DS, R-11, R-12

DSKCHG, [R-12](#)
ECM, [M-38](#)
EN018B, [L-26](#)
ENTEREXIT, [G-30](#)
ENV2ATTDUR, [N-3](#)
ENV2DECDUR, [N-3](#)
ENV2RELDUR, [N-3](#)
ENV2SUSDUR, [N-3](#)
ENV3ATTDUR, [N-3](#)
ENV3DECDUR, [N-3](#)
ENV3OUT, [N-4](#)
ENV3RELDUR, [N-3](#)
ENV3SUSDUR, [N-3](#)
ENVXATTDUR, [N-3, N-4](#)
ENVXDECDUR, [N-3, N-4](#)
ENVXRELDUR, [N-3, N-4](#)
ENVXSUSDUR, [N-3, N-4](#)
EQ, [R-12](#)
ETRIG, [L-23, L-26](#)
ETRIGMAPD, [L-23, L-26](#)
EV1, [R-22](#)
EV2, [R-22](#)
EXGLYPH, [M-44](#)
EXSID, [G-30](#)
EXTIROS, [M-44](#)
EXTSYNC, [M-41](#)
F4502, [G-30](#)
FAST, [M-41](#)
FCLRHI, [M-44](#)
FCLRLO, [M-44](#)
FDC2XSEL, [R-20](#)
FDCENC, [R-19, R-20](#)
FDCTIBEN, [R-20](#)
FDCVARSPD, [R-20](#)
FILLVAL, [R-19, R-20](#)
FLG, [O-3, O-5](#)
FLTRBDPASS, [N-4](#)
FLTRCUTFRQHI, [N-3, N-4](#)
FLTRCUTFRQLO, [N-3, N-4](#)
FLTRCUTV3, [N-4](#)
FLTREXTINP, [N-4](#)
FLTRHIPASS, [N-4](#)
FLTRLOPASS, [N-4](#)
FLTRRESON, [N-3, N-4](#)
FLTRVOL, [N-3, N-4](#)
FLTRVXOUT, [N-4](#)
FNRASTERLSB, [M-42, M-44](#)
FNRASTERMSB, [M-42, M-44](#)
FNRST, [M-44](#)
FNRSTCMP, [M-45](#)
FRAMECOUNT, [G-18, G-19](#)
FREE, [R-12](#)
FRMERR, [P-3](#)
GEORAMBASE, [G-30](#)
GEORAMMASK, [G-30](#)
GESTUREDIR, [R-22](#)
GESTUREID, [R-22](#)
H1280, [M-41](#)
H640, [M-41](#)
HDSCL, [P-5](#)
HDSDA, [P-5](#)
HICKED, [G-30](#)
HOTREG, [M-45](#)
HPOS, [M-40, M-41](#)
HSYNCP, [M-45](#)
HTRAP01, [G-17](#)
HTRAP02, [G-17](#)
HTRAP03, [G-17](#)
HTRAP04, [G-17](#)
HTRAP05, [G-17](#)
HTRAP06, [G-17](#)
HTRAP07, [G-17](#)
HTRAP08, [G-17](#)
HTRAP09, [G-17](#)
HTRAP0A, [G-17](#)
HTRAP0B, [G-17](#)
HTRAP0C, [G-17](#)
HTRAP0D, [G-17](#)
HTRAP0E, [G-17](#)
HTRAP0F, [G-17](#)
HTRAP10, [G-17](#)
HTRAP11, [G-17](#)
HTRAP12, [G-17](#)
HTRAP13, [G-17](#)

HTRAP14, G-17
HTRAP15, G-17
HTRAP16, G-17
HTRAP17, G-17
HTRAP18, G-17
HTRAP19, G-17
HTRAP1A, G-17
HTRAP1B, G-17
HTRAP1C, G-17
HTRAP1D, G-17
HTRAP1E, G-17
HTRAP1F, G-17
HTRAP20, G-17
HTRAP21, G-17
HTRAP22, G-18
HTRAP23, G-18
HTRAP24, G-18
HTRAP25, G-18
HTRAP26, G-18
HTRAP27, G-18
HTRAP28, G-18
HTRAP29, G-18
HTRAP2A, G-18
HTRAP2B, G-18
HTRAP2C, G-18
HTRAP2D, G-18
HTRAP2E, G-18
HTRAP2F, G-18
HTRAP30, G-18
HTRAP31, G-18
HTRAP32, G-18
HTRAP33, G-18
HTRAP34, G-18
HTRAP35, G-18
HTRAP36, G-18
HTRAP37, G-18
HTRAP38, G-18
HTRAP39, G-18
HTRAP3A, G-18
HTRAP3B, G-18
HTRAP3C, G-18
HTRAP3D, G-18
HTRAP3E, G-18
HTRAP3F, G-18
HTRAPXX, G-17, G-19
IFRXIRQ, P-3
IFRXNMI, P-3
IFTXIRQ, P-3
IFTXNMI, P-3
ILP, M-38
IMALRM, O-7, O-8
IMFLG, O-7, O-8
IMODA, O-3, O-5
IMODB, O-3, O-5
IMRXIRQ, P-3
IMRXNMI, P-3
IMSP, O-7, O-8
IMTB, O-7, O-9
IMTXIRQ, P-3
IMTXNMI, P-3
INDEX, R-12
INT, M-41
IR, O-3, O-5
IRQ, R-12
ISBC, M-38
ISRCLR, O-3, O-5
ISSC, M-38
J21H, P-4, P-5
J21HDDR, P-4, P-5
J21L, P-4, P-5
J21LDDR, P-4, P-5
JMP32EN, G-30
JOYSWAP, P-5
KEY, M-40-M-42, M-45
KEYLEDEN, P-5
KEYLEDREG, P-4, P-5
KEYLEDVAL, P-4, P-5
KEYLEFT, P-5
KEYUP, P-5
KSCNRATE, P-4, P-5
LATCHINT, G-23
LED, R-12
LINESTEPLSB, M-43, M-45
LINESTEPMSB, M-43, M-45

LJOYA, P-5
LJOYB, P-5
LOAD, O-3, O-5
LOST, R-12
LPX, M-37, M-38
LPY, M-37, M-38
M65MODEL, P-5
MACADDR2, Q-9
MACADDR3, Q-9
MACADDR4, Q-9
MACADDR5, Q-9
MACADDR6, Q-9
MACADDRX, Q-9
MALT, P-5
MAPEDPAL, M-43, M-45
MAPHI, G-29, G-30
MAPHIMB, G-29, G-30
MAPLO, G-29, G-30
MAPLOMB, G-29, G-30
MATHINO, G-20
MATHIN1, G-20
MATHIN2, G-20, G-21
MATHIN3, G-21
MATHIN4, G-21
MATHIN5, G-21
MATHIN6, G-21
MATHIN7, G-21
MATHIN8, G-21
MATHIN9, G-21
MATHINA, G-21
MATHINB, G-21
MATHINC, G-22
MATHIND, G-22
MATHINE, G-22
MATHINF, G-22
MATHINX, G-20, G-23
MATRIXEN, G-31
MC1, M-37, M-38, M-40-M-42,
 M-45
MC2, M-37, M-38, M-40-M-42,
 M-45
MC3, M-38, M-40-M-42, M-45
MCAPS, P-5
MCM, M-38
MCST, Q-10
MCTRL, P-6
MDISABLE, P-6
MIIMPHY, Q-9, Q-10
MIIMREG, Q-9, Q-10
MIIMVLSB, Q-9, Q-10
MIIMVMSB, Q-9, Q-10
MISBC, M-39
MISSC, M-39
MIXREGDATA, R-25, R-26
MIXREGSEL, R-25, R-26
MLSHFT, P-6
MMEGA, P-6
MONO, M-41
MOTOR, R-12
MRIRQ, M-39
MRSHTF, P-6
MSCRL, P-6
MULBUSY, G-23
MULTINA, G-20, G-23
MULTINB, G-20, G-24
MULTOUT, G-20, G-24
NOBUF, R-12
NOCRC, Q-10
NOEXROM, G-19
NOGAME, G-19
NOMIX, L-26
NOPROM, Q-10
NORRDEL, M-45
OCEANA, G-19
OMODA, O-3, O-5
OMODB, O-3, O-5
OSC3RNG, N-3, N-4
OSKALT, P-6
OSKDEBUG, P-6
OSKDIM, P-6
OSKEN, P-6
OSKTOP, P-6
OSKZEN, P-6
OSKZON, P-6

PADDLE1, [N-3](#), [N-4](#)
PADDLE2, [N-3](#), [N-4](#)
PAL, [M-41](#)
PALBLUE, [M-40](#), [M-41](#)
PALEMU, [M-45](#)
PALGREEN, [M-40](#), [M-41](#)
PALNTSC, [M-45](#)
PALRED, [M-40](#), [M-41](#)
PBONA, [O-4](#), [O-5](#)
PBONB, [O-4](#), [O-5](#)
PCH, [G-29](#), [G-31](#)
PCL, [G-29](#), [G-31](#)
PCODE, [R-11](#), [R-12](#)
PETSCIKEY, [P-4](#), [P-6](#)
PFLAGS, [G-29](#), [G-31](#)
PIRQ, [G-31](#)
PNMI, [G-31](#)
PORT00, [G-29](#), [G-31](#)
PORT01, [G-29](#), [G-31](#)
PORTA, [O-3](#)-[O-5](#)
PORTB, [O-3](#)-[O-5](#)
PORTF, [P-4](#), [P-6](#)
PORTFDDR, [P-4](#), [P-6](#)
POTAX, [P-4](#), [P-6](#)
POTAY, [P-4](#), [P-6](#)
POTBX, [P-4](#), [P-6](#)
POTBY, [P-4](#), [P-6](#)
POWEREN, [G-19](#)
PREFETCH, [G-19](#)
PROT, [R-12](#)
PTYEN, [P-3](#)
PTYERR, [P-3](#)
PTYEVEN, [P-3](#)
PWMPDM, [R-26](#)
RAND, [G-18](#), [G-19](#)
RASCMP, [M-43](#), [M-45](#)
RASCMPMSB, [M-43](#), [M-45](#)
RASLINE0, [M-43](#), [M-45](#)
RASTERHEIGHT, [M-43](#), [M-45](#)
RC, [M-37](#), [M-39](#)
RC8, [M-39](#)
RCENABLED, [Q-10](#)
RDCMD, [R-12](#)
RDREQ, [R-12](#)
READBACKLSB, [R-25](#), [R-26](#)
READBACKMSB, [R-25](#), [R-26](#)
REALHW, [P-6](#)
REGA, [G-29](#), [G-31](#)
REGB, [G-29](#), [G-31](#)
REGX, [G-29](#), [G-31](#)
REGZ, [G-29](#), [G-31](#)
RESERVED, [G-23](#), [G-24](#)
RESV, [M-43](#), [M-45](#)
IRQ, [M-39](#)
RMODA, [O-4](#), [O-5](#)
RMOdB, [O-4](#), [O-5](#)
RNF, [R-12](#)
ROM8, [M-41](#)
ROMA, [M-41](#)
ROMC, [M-41](#)
ROME, [M-42](#)
ROMPROT, [G-31](#)
RSEL, [M-39](#)
RST, [M-39](#), [Q-10](#)
RST41, [P-6](#)
RSTDELEN, [M-45](#)
RSVD, [G-31](#)
RUN, [R-12](#)
RXBF, [Q-9](#), [Q-10](#)
RXBLKD, [Q-10](#)
RXEN, [P-3](#)
RXOVRRUN, [P-3](#)
RXPH, [Q-9](#), [Q-10](#)
RXQ, [Q-10](#)
RXQEN, [Q-10](#)
RXRDY, [P-3](#)
S1X, [M-37](#)
S1Y, [M-37](#)
S2X, [M-37](#)
S2Y, [M-37](#)
S3X, [M-37](#)
S3Y, [M-37](#)
S4X, [M-37](#)
S4Y, [M-37](#)

S5X, [M-37](#)
S5Y, [M-37](#)
S6X, [M-37](#)
S6Y, [M-37](#)
S7X, [M-37](#)
S7Y, [M-37](#)
SBC, [M-37](#), [M-39](#)
SCM, [M-37](#), [M-39](#)
SCREENCOL, [M-37](#), [M-39](#), [M-40](#),
 [M-42](#), [M-45](#)
SCRNPTRBNK, [M-43](#), [M-45](#)
SCRNPTRLSB, [M-43](#), [M-45](#)
SCRNPTRMB, [M-43](#), [M-45](#)
SCRNPTRMSB, [M-43](#), [M-45](#)
SDBDRWDLSB, [M-43](#), [M-45](#)
SDBDRWDMBSB, [M-43](#), [M-45](#)
SDBSH, [P-6](#)
SDCLK, [P-6](#)
SDCS, [P-7](#)
SDDATA, [P-7](#)
SDR, [O-3](#)-[O-6](#)
SE, [M-37](#), [M-39](#)
SECTOR, [R-11](#), [R-12](#)
SECTOR0, [R-19](#), [R-20](#)
SECTOR1, [R-19](#), [R-20](#)
SECTOR2, [R-19](#), [R-20](#)
SECTOR3, [R-19](#), [R-20](#)
SEXX, [M-37](#), [M-39](#)
SEXY, [M-37](#), [M-39](#)
SHDEMU, [M-46](#)
SIDE, [R-11](#), [R-12](#)
SIDMODE, [N-4](#)
SILENT, [R-15](#)
SLIEN, [G-19](#)
SMTH, [M-46](#)
SNX, [M-37](#), [M-39](#)
SNY, [M-37](#), [M-39](#)
SP, [O-4](#), [O-6](#)
SPH, [G-29](#), [G-31](#)
SPL, [G-29](#), [G-31](#)
SPMOD, [O-4](#), [O-6](#)
SPR16EN, [M-43](#), [M-46](#)
SPR1COL, [M-38](#)
SPR2COL, [M-38](#)
SPR3COL, [M-38](#)
SPR4COL, [M-38](#)
SPR5COL, [M-38](#)
SPR6COL, [M-38](#)
SPR7COL, [M-38](#)
SPRALPHAVAL, [M-43](#), [M-46](#)
SPRBPMEN, [M-42](#), [M-46](#)
SPRENALPHA, [M-43](#), [M-46](#)
SPRENV400, [M-43](#), [M-46](#)
SPRH640, [M-46](#)
SPRHGHT, [M-42](#), [M-46](#)
SPRHGTEN, [M-42](#), [M-46](#)
SPRMC0, [M-38](#)-[M-40](#), [M-42](#),
 [M-46](#)
SPRMC1, [M-38](#)-[M-40](#), [M-42](#),
 [M-46](#)
SPRNCOL, [M-38](#), [M-39](#)
SPRPALSEL, [M-43](#), [M-46](#)
SPRPT16, [M-46](#)
SPRPTRADRLSB, [M-43](#), [M-46](#)
SPRPTRADRMSB, [M-43](#), [M-46](#)
SPRPTRBANK, [M-43](#), [M-46](#)
SPRTILEN, [M-42](#), [M-46](#)
SPRX64EN, [M-42](#), [M-46](#)
SPRXSMSBS, [M-43](#), [M-46](#)
SPRYADJ, [M-43](#), [M-46](#)
SPRYSMSBS, [M-43](#), [M-46](#)
SPTRCONT, [M-46](#)
SSC, [M-37](#), [M-39](#)
STEP, [R-11](#), [R-12](#)
STRM, [Q-10](#)
STRTA, [O-4](#), [O-6](#)
STRTB, [O-4](#), [O-6](#)
SWAP, [R-12](#)
SXMSB, [M-37](#), [M-39](#)
SYNCMOD, [P-3](#), [P-4](#)
SYSCTL, [P-4](#), [P-7](#)
TA, [O-4](#), [O-6](#)
TALATCH, [O-6](#)-[O-9](#)

TARGANY, R-16
TB, O-4, O-6
TBDRPOS, M-42, M-46
TEXTXPOS, M-42, M-46
TEXTYPOS, M-42, M-47
TIMERA, O-3-O-6
TIMERB, O-3-O-6
TK0, R-12
TOD50, O-4, O-6
TODAMPM, O-4, O-6, O-7, O-9
TODEDIT, O-4, O-6
TODHOUR, O-3-O-9
TODJIF, O-3-O-9
TODMIN, O-3, O-4, O-7-O-9
TODSEC, O-3-O-9
TOUCH1XLSB, R-22
TOUCH1XMSB, R-22
TOUCH1YLSB, R-22
TOUCH1YMSB, R-22
TOUCH2XLSB, R-22
TOUCH2XMSB, R-22
TOUCH2YLSB, R-22, R-23
TOUCH2YMSB, R-22, R-23
TRACK, R-11, R-13
TXEN, P-4
TXIDLE, Q-10
TXPH, Q-9, Q-10
TXQ, Q-10
TXQEN, Q-10
TXRST, Q-10
TXSZLSB, Q-9, Q-10
TXSZMSB, Q-9, Q-10
UARTDATA, G-30, G-31
UFAST, P-7
UNIT1INA, G-22
UNIT1INB, G-22
UNIT1OUT, G-22
UNIT2INA, G-22
UNIT2INB, G-22
UNIT2OUT, G-22
UNIT3INA, G-22
UNIT3INB, G-22
UNIT3OUT, G-23
UNIT4INA, G-22
UNIT4INB, G-22
UNIT4OUT, G-23
UNIT5INA, G-22
UNIT5INB, G-22
UNIT5OUT, G-23
UNIT6INA, G-22
UNIT6INB, G-22
UNIT6OUT, G-23
UNIT7INA, G-22
UNIT7INB, G-22
UNIT7OUT, G-23
UNIT8INA, G-22
UNIT8INB, G-22
UNIT8OUT, G-23
UNIT9INA, G-22
UNIT9INB, G-22
UNIT9OUT, G-23
UNITAINA, G-22
UNITAINB, G-22
UNITAOUT, G-23
UNITBINA, G-22
UNITBINB, G-22
UNITBOUT, G-23
UNITCINA, G-22
UNITCINB, G-22
UNITCOUT, G-23
UNITDINA, G-22
UNITDINB, G-22
UNITDOUT, G-23
UNITEINA, G-22
UNITEINB, G-22
UNITEOUT, G-23
UNITFINA, G-22
UNITFINB, G-22
UNITFOUT, G-23
UNITXINA, G-22, G-24
UNITXINB, G-22, G-24
UNITXOUT, G-22, G-24
UPDN1, R-21, R-23
UPDN2, R-21, R-23

USEREAL0, R-16
USEREAL1, R-16
UXBSADD, G-24
UXDVADD, G-24
UXHIOUT, G-24
UXLATCH, G-24
UXLOWOUT, G-24
UXMLADD, G-24
V400, M-42
VDRQ, R-20
VEQINH, R-20
VFAST, M-47
VFDC0, R-20
VFDC1, R-20
VFLOP, G-31
VGAHDTV, M-47
VICIII, R-20
VICMODE, G-29, G-31
VIRTKEY1, P-4, P-7
VIRTKEY2, P-4, P-7
VIRTKEY3, P-4, P-7
VLOST, R-20
VOICE1CTRLRMF, N-4
VOICE1CTRLRMO, N-4
VOICE2CTRLRMF, N-4
VOICE2CTRLRMO, N-4
VOICE2FRQHI, N-3
VOICE2FRQLO, N-3
VOICE2PWHI, N-3
VOICE2PWLO, N-3
VOICE2UNSD, N-3
VOICE3CTRLRMF, N-4
VOICE3CTRLRMO, N-4
VOICE3FRQHI, N-3
VOICE3FRQLO, N-3
VOICE3PWHI, N-3
VOICE3PWLO, N-3
VOICE3UNSD, N-3
VOICEXCTRLGATE, N-5
VOICEXTRLPUL, N-5
VOICEXCTRLRNW, N-5
VOICEXCTRLSAW, N-5
VOICEXCTRLTRI, N-5
VOICEXCTRLTST, N-5
VOICEXFRQHI, N-3, N-5
VOICEXFRQLO, N-3, N-5
VOICEXPWHI, N-3, N-5
VOICEXPWLO, N-3, N-5
VOICEXUNSD, N-3, N-5
VPOS, M-40, M-42
VRFOUND, R-20
VRNF, R-20
VS, M-37, M-39
VSYNCNP, M-47
VWFOUND, R-20
WATCHDOG, G-30, G-31
WGATE, R-13
WRCMD, R-13
WREN, G-24
WTREQ, R-13
XINV, R-23
XPOSLSB, M-42, M-47
XPOSMSB, M-42, M-47
XSCL, M-37, M-39
YINV, R-23
YSCL, M-37, M-39
relational operators, 8-30
RENUMBER, 8-38
RESQ, H-139
revers, 17-7
RLA, H-45
RMB0, H-104
RMB1, H-105
RMB2, H-105
RMB3, H-105
RMB4, H-106
RMB5, H-106
RMB6, H-107
RMB7, H-107
RND(), 8-45
ROL, H-45, H-107
ROLQ, G-14, H-140
Root directory, J-12
ROR, H-46, H-108

RORQ, G-14, H-140
ROW, H-109
RRA, H-47
RSVQ, H-141
RTI, H-47, H-110
RTS, H-48, H-110
RXNORMAL, Q-11
RXONLYONE, Q-11

SAVE, K-13
SAX, H-48
SBC, H-49, H-50, H-110, H-142
SBCQ, G-14, H-143
SBX, H-50
scientific notation, 8-45
Screen memory and colour RAM, B-8
SD Cards
 Formatting, 4-3
 Locations, 2-7
SEC, H-51, H-111
SED, H-51, H-112
SEE, H-112
SEI, H-51, H-113
set16bitcharmode, 17-6
setcharsetaddr, 17-4
setcolramoffset, 17-4
setextendedattrib, 17-6
sethotregs, 17-6
setmapedpal, 17-10
SETMEMORY, K-22
setpalbank, 17-9
setpalbanka, 17-9
setpalentry, 17-10
SETPC, K-20
setscreenaddr, 17-3
setscreensize, 17-5
SHA, H-52
SHX, H-52
SHY, H-53
SLO, H-53
SMB0, H-113
SMB1, H-114
SMB2, H-114

SMB3, H-114
SMB4, H-115
SMB5, H-115
SMB6, H-115
SMB7, H-116
SRE, H-54
STA, H-55, H-116, H-144
STARTTX, Q-11
STEP, 8-13
STOP, 8-39
STOPTX, Q-11
STQ, G-14, H-144
string, 8-13
STX, H-55, H-116
STY, H-55, H-117
STZ, H-117
SYNTAX ERROR, 8-4

TAB, H-118
TAS, H-56
TAX, H-56, H-118
TAY, H-57, H-119
TAZ, H-119
TBA, H-120
textcolor, 17-7
Texture Scaling, L-17
THEN, 8-30
TIB, R-8, R-11
togglecase, 17-6, 17-7
TRACE, K-22
Track Information Block, R-8, R-11
TRANSFER, K-13
TRB, H-120
TSB, H-121
TSX, H-57, H-122
TSY, H-122
TXA, H-58, H-123
TXS, H-58, H-123
TYA, H-59, H-123
Type mismatch error, 8-17
TYS, H-124
TZA, H-124

UARTDIVISOR, [K-18](#)

underline, [17-8](#)

unequal, [8-31](#)

Utility Menu, [I-5](#)

variable, [8-8](#)

 numeric, [8-15](#)

 string, [8-15](#)

VERIFY, [K-13](#)

vline, [17-12](#)

Warnings

 Extra Ignored, [8-22](#)

WATCHPOINT, [K-22](#)

wherex, [17-14](#)

wherey, [17-15](#)

XAA, [H-59](#)

About the MEGA65 Book

C64 and C65 program and peripheral compatibility ... amazing sound ... true arcade-class graphics ... beautifully finished hardware ... full mechanical keyboard ... rich networking capabilities and one of the fastest 6502-class processors available make the MEGA65 truly unique for home, business or educational use.

The MEGA65 Complete Compendium collects into a single huge volume all the information you need to know about your MEGA65. Collecting all of the key information of the various MEGA65 user's guides and reference manuals in once place, this book provides detailed information on every topic, including BASIC programming, sound, graphics, networking, assembly language programming, cross-platform development, including using high-level languages like C or KickC, the MEGA65's powerful 8-bit chipset, and how to use its powerful FPGA core to implement other computer systems.

This book is the must-have reference for the MEGA65, that every user, whether beginner or advanced, should have with them when exploring the full potential of the MEGA65.

With authors including professional writers, university lecturers and 8-bit experts, the content is both accessible and extensive. Beginners will find detailed explanations of how to get started, while advanced users will find highly detailed technical information on the inner workings of every aspect of the MEGA65.

In short, this book is designed for you to get the most out of the MEGA65's extensive capabilities.

