Writing Platform Independent Code on CBM Machines

Table of Contents

Introduction

There exists a considerable number of different types and revisions of 8-bit Commodore (aka CBM) machines. Back in the day, when we had only very few – and sometimes also very poorly written or erroneous – descriptions and limited experience on these computers, they seemed rather different from each other, and it consequently also seemed then not that much possible to write a program which could be run on all of them (or actually any random two or three of them at least).

However, they are not so different in real: the main architecture of the CPU's and their machine languages are mostly the same (apart from the illegal opcodes and some other extensions), as well as the core of Basic dialects, the memory organization and the most important Kernal calls etc. So thus it ***is*** possible to write such code; although it will not be too easy, and requires some knowledge about all of them and a very careful, quirky and neat design to fit everywhere… Since I have started to develop my *Rosetta Interactive Fiction* project in the last few years (2012-2017 when writing this; *plus some 2020 fixes, see them at the very end, whereas being referred to as* ***+note:***(…)*** *inside the text*) in my sparetime, I have slowly awoken to this, and decided to make it so. And then decided to also write this little guide (or programming handbook) formed out of my gained experiences, for making it much easier to others (who would later also decide to do so and begin this way).

As a first step here, let us see the competitors, one by one:

C64

It has always 64K memory, and the 6510 CPU has two built-in I/O ports at $00 and $01 for paging some areas within (starting from $a000). The screen memory is at $0400 (and the colour memory at $d800), below which are system areas. The Basic memory starts at $0801 (2049), and a single file can even be loaded up to $d000. There are many different Kernal ROM's (e.g. one for JiffyDOS), yet the last two bytes of them are always $ff48 (the jumping vector of the IRQ). Some important system variables can be found at $2b-$2c (43-44) and $2d-$2e (45-46) for the Basic program starting and ending (and the starting of the Basic variables as well). These can be used for identification.

C64 (Extensions and Clones or Variants)

SuperCPU: an external card with 65816 CPU and onboard memory. The instruction set is quite different (no illegal opcodes can be used), with support for 24-bit native memory handling (up to 16 MB) and clock be switched over between 1 and 20 MHz. Some useful locations are at $d07a and $d07b (writing to here turns the turbo off and on), $d0b8 (the actual state of turbo setting can be read) or $d27b-$d27f (some parameters of the memory).

Flash 8, Turbo Process and Turbo Master (aka Schnedler card): similar beasts, the former two use the same 65816 CPU at about 8 and 4 MHz, while the latter uses a 65C02 at 4 MHz instead. All of these cards have similar (but actually different) switchover bits (either bit6 or bit7, i.e. $40 or $80) at either $00 or $01 in memory, and sometimes in reverse (so sometimes the set, while otherwise the reset state of the bit signs the turbo being on and off).

TDC (or Turbo Demo Card): an overclocking daughtercard for HMOS motherboards. It has two bits at $d030 for 2 and 4 MHz turbo modes (similarly to the C128). $d02f is supposed to contain $f0 (actually $ff in real) and $d031 a version number (usually $d1 in real). While in turbo mode, it accelerates everything, except for the VIC-II and SID: not only the CPU, but also including the CIA's (so the interrupts get thicker and the cursor faster, and the disk operations not possible) and the memory, plus it has some other quirks.

C64DTV (Direct-To-TV): also has a hidden turbo mode (if activated, the disk operations are not possible and there are no VIC-II badlines) and some RAM (up to 2 MB) accessible via DMA (controller at $d300), plus a blitter and screen modes (256 colours). A set of hidden registers can be made visible by writing $01 to $d03f (and invisible by $00).

Turbo Chameleon: a flexible turbo mode from 2 even up to 14 MHz (be switched at $d030 like the C128 and TDC) and the widest scale of further expansions built in. Can be detected by writing $2a to $d0fe and reading back (any other value different from $ff which must normally be there), then writing $ff to there and reading back again (it must already give back $ff then). The first attempt makes visible, while the second invisible its hidden registers.

1541 Ultimate (I, II or II+) or U64 +note:(1)***: if the Command Interface is enabled, then it can be detected by reading from $df1d which must give back $c9 then.

IDE64: the three bytes from $de60 through $de62 must contain the values of $49, $44 and $45 (the letters of "ide").

Emulators: some C64 emulators (like the C64S and the v2.2 or older versions of VICE) make it possible to detect if it is not real hardware. The $dfff memory location at the I/O area oscillates between $55 and $aa at each and every read attempt in this case (some further and more advanced emulator detection techniques will be discussed later, too).

Memory expansions: like REU or GeoRAM etc. (See the user manual of my independent *MemTest64* project for them!)

C128

The big brother with 128K or 256K memory (in two or four 64K banks) and 8502 CPU (switchable between its 1 and 2 MHz modes at $d030 or via the Basic commands SLOW and FAST), the VIC-IIe (a slightly modified VIC-II) and VDC video chips (with 16K or 64K separated video memory) and an additional Z80A processor (at effectively 2 MHz, too). The screen and colour memories are as the same as those of the C64 by default, while the last two bytes of Kernal are always $ff17. The Basic area starts from $1c01 (7169) and lasts till $feff. The system variables of the Basic starting can be found at $2d-$2e (instead of $2b-$2c which are not really used and generally may contain just zeroes or any other meaningless values). The Basic variables are stored in the second bank. (The VDC is accessible at $d600-$d601.)

There are also some system variable areas from $0800 up to $12ff. From $1300 to $1bff the memory is not used (it is called as Application Program Area and free), so thus the Basic starting might even be set as low as this at any time as well.

The targeted memory bank (for PEEK and POKE) is selected by the BANK command: from BANK 0 to BANK 3, and also BANK 15 (the system bank, which basically equals to the first bank, but also with I/O and ROM areas mapped in, from $4000 onwards). The bottom 1K and the uppermost 256 bytes are common memory (included in every bank). In machine language, the $ff00 location is used for bank switching: the corresponding values of $3f/$7f/$bf/$ff do the same as BANK 0/1/2/3, and the value of $00 the same as BANK 15.

You can switch to C64 mode by typing in the GO64 command (or by holding down the C= key at startup, or via the JMP $FF4D Kernal call). In this mode only the first bank can be accessed (and it also keeps its whole content from after the native mode except for the overwritten bytes). The VDC stays also accessible in this mode, as well as the locations at $d02f (has three bits for scanning the extended keyboard) and $d030 (has two bits, one for the fast mode and the other for the test bit). So these can be used for identification in C64 mode.

When the 2 MHz fast mode is activated, the VIC-IIe screen seems garbled, so it must also be blanked. (Still you may apply the common trick by making two raster interrupts at the bottom and the top of the visible screen area, and turning it on and off, thus achieving some average of 30% speed gain over the normal operating mode.)

C65 (or C64DX Prototypes)

This “yet even more” big brother has 128K memory in two 64K banks, and a 4510 CPU (which is a 65CE02 variant in real with different instruction set and some slightly faster timings that make it nearly 20% faster in advance, on top of that it can be switched over between its 1 and 3.5 MHz slow and fast modes, too), and a VIC-III (which has got no badlines, it makes it even faster, plus an integrated 80-column mode). The screen memory is at $0800 by default, and the Basic starting is at $2001 (8193). The system variables of the latter are also stored at $2d-$2e (as the same as those of the C128), and it also has a similar free memory block from $1300 to $1fff (so the Basic area may be expanded to here, too).

The FAST and SLOW commands basically work the same (just without the screen blanking and considering that the fast mode is default), as well as those BANK 0 and BANK 1, however the system bank is BANK 128 here. (The memory might be expanded even up to 8 MB, thus up to BANK 127 might also be used.) BANK 2 and 3 are for the ROM’s.

The last two bytes of Kernal may vary, since there are many different “half-made” (prototyped) revisions of them: at least six (or more) can be found on the internet, which have got these following values there: $fab1, $fb6a, $fb75, $fb80, $fb84 and $fb87.

Turning into C64 mode goes similarly, more or less (the Kernal call is JMP $FF53 here). The CPU is switched to 1 MHz mode in this case, otherwise its behaviour is not changed (the instruction set and the faster timing, and also the badlines) ***+note:***(2)***, so the real C64 compatibility will be rather limited just because of this. The extended registers of VIC-III stay hidden until you write two “magic bytes”: first $a5, then $96 into the location at $d02f (writing any other value makes them being hidden again).

The machine has got an integrated DMA controller at $d700 (also hidden by default), which is somewhat similar to that of the REU (but much more capable). The bank switching method in Assembly is rather complicated (including a mapper function of the CPU).

C65 Emulator (MESS)

This is the only (more or less) usable emulation of the original machine(s) up to here I know. Several things work slightly differently though (man needs to be aware of which):

The $00 and $01 memory locations seem to do nothing here (in spite of that these ports must have been both to be used in C64 and C65 modes). Thus, some memory areas just cannot be moved away, and not even in C64 mode: the Basic, the Kernal and the I/O will always remain there, so the RAM under them cannot be accessed.

Any writing into $d02f gains no effect, either (neither of the "magic bytes" nor any other value): so the "hidden" registers are also always there.

The emulations always use an NTSC hardware (even if it is reported as "PAL").

In the C64 mode, the cursor is blinking (and the keys are repeating) way faster than normal (no idea why). On the real hardware, it works the same as on a C64 (here not).

If you would like to also try it, then the *MESS* ***v0.111*** version (from 2006) will be needed (and possibly its GUI extension, called MessGui, too). This is necessary unfortunately just because all newer versions of MESS have got their entire C65 emulations broken (thus totally useless). Once having the right version, it is recommended to be started with the *v0.9.**910111*** (i.e. '91. January) Kernal ROM version (that originates from the earlier and probably the most common *Rev2B* motherboard); just because that same binary is used by the C65GS, too. (The MESS version has also got a lot of other flaws and quirks, and because of them the internal drive with device number 8 is not so usable. So must you then attach a D64 image as an external drive with 10 or 11, and start it all from there.)

Also, resetting the emulation is not always working properly (so you better quit and start it again if you need reset).

MEGA65 (or C65GS)

An FPGA re-implementation (or simply called a modern clone) of both the C65 and the C64 platforms (and both of them with better compatibility as well as with further feature sets). This has not yet been released when writing this, still already in the making for some years (2013-2020), so any of the information provided here might be wrong:

It has got the same 128K main memory as the C65 prototypes, and over that some expanded memory, too (16 MB or 256 MB or even more, depending on the actual FPGA board being used). It can be built around a Nexys 4 or a custom-made own motherboard.

It has got a brand-new VIC-IV video chip implementation: including the feature set of VIC-III (which can be activated by using the same above "magic bytes"), and on top of that, an even much more powerful new feature set – which can be activated by using two other "magic bytes": $47 and $53 written into $d02f.

The C64 mode is promised to will have a far higher grade of backwards compatibility to the original platform: the instruction set be switched over to the 6510 one (with all the undocumented opcodes, too), as well as the timing be so accurate (including the badlines).

Both C65/C64 modes to have some newer fast and turbo speeds: on top of the former normal 3.5 MHz, also 2 MHz (switchable at $d030 like the Turbo Chameleon and the C128) and a maximum of 40 MHz (or even more): can be used as POKE 0,65 and POKE 0,64.

Any (or many) of the already existing C65 and C64 Kernal ROM's might be used.

And so on… Please see the complete and more actual information page here:

http://mega65.org/

VIC-20

And this is the elder little brother of all the above ones. The system ROM's (both the Kernal and the Basic ones) are very similar to (or almost the same as) those of the C64. The last two bytes are $ff72 though. However, there are many different memory configurations possible. The unexpanded stock machine has only 5K memory: 1K for the system variable area, and 4K starting at $1000 (so thus there is a "hole" in-between, from $0400 to $0fff in this case). The 8K machines fill in this hole with another 3K added. And then it can be further expanded up to 32K (up to $7fff) as a contiguous space, and even with another 8K (from $a000 to $bfff) apart. So it is a 40K in total.

The Basic starting pointer is at $2b-$2c (and the ending pointer is at $2d-$2e), but there are three variations for the default Basic start: $1001 (4097) on a 5K (stock) machine, $0401 (1025) on 8K, and $1201 (4609) otherwise. There are also two variations for the default screen (and colour) data: either at $1e00 (colours at $9600) on the 5K/8K, or at $1000 (colours at $9400) otherwise.

The smaller configs can be "simulated" on a fully expanded machine (by manually being set), which is often needed when you want to run a program made (or optimized) for the other memory configs. Not only the Basic pointers must be altered in this case, but also the high byte of the screen at $288 (648), and then after having that finished, even done a Kernal init call by JSR $E518 (or SYS 58648), before performing the LOAD and RUN commands.

Its 6502 CPU runs slightly faster than that of the C64 (1.1 MHz on PAL or 1 MHz on NTSC), and the VIC-I has no badlines (as neither can be blanked). The screen size is 22x23 characters, which stays under 0.5K after all.

TED Series

Their 7501 CPU may have got a full speed at about 1.7 MHz (on screen blanked), but the TED video chip reduces that only to about 1.1 MHz in average (by clocking it down to 0.89 MHz while within the visible screen area and turning back to 1.7 MHz on borders; which is somewhat similar to the software trick on C128, but here is being done by hardware).

Their memory can be 16K (C16 or C116), 32K (C232 prototype) or 64K (Plus/4). If it is not fully expanded, then the physical memory is also mirrored upwards in the address space (that means e.g. on a C16 the RAM location at $1000 equals to the locations at $5000, $9000 and $d000). The Basic starts at $1001 (4097), and the pointers are the same as on the C64 and VIC-20. The screen memory is at $0c00 (and the colour memory at $0800). The Kernal and the Basic can be paged into the upper 32K address space in Assembly by writing anything to $ff3e, and be paged out by writing anything to $ff3f.

The last two bytes of Kernal are $fcb3, but it can be only read in Assembly, because in Basic the PEEK command reads from RAM instead.

LCD (Prototype)

65C02 (or 65C102) CPU at 1 or 2 MHz, with 32K or 64K memory. The Basic is the most similar to that of the TED series (v3.6 vs v3.5), also starting at $1001 (4097), but the pointers are stored at $65-$66 (101-102). The screen memory is at $0800 (with no colours because it is monochrome), organized into 16 rows of 128 bytes size, of which only 80 bytes per line are used (80x16 characters). The last two bytes of Kernal are $fa0e here.

PET Series

6502 CPU at 1 MHz, from 4K to 32K contiguous memory, plus another 8K possible apart (like on VIC-20, but from $9000 to $afff instead). The Basic starts at $0401 (1025), the pointers at $28-$29 (40-41). The screen memory is at $8000 (either 1K or 2K depending on actual size of 40x25 or 80x25 in monochrome). The last two bytes of Kernal may vary among three values (depending on version): $e66b (rev. 1-2), $e61b (rev. 3) and $e442 (rev. 4).

CBM-II Series

Their rather special 6509 CPU is able to handle up to 1 MB memory via bank switching. The memory is organized in 64K banks, so thus up to 16 banks are possible. While it sounds well at first, it is even worse in real: just because the switching method is so primitive, that either the whole address space is switched over (including the zero page, the stack and even the running code itself), or we can be only cueing slowly, byte by byte. The CPU has two I/O ports at $00 and $01, each of which can hold a value between 0 and 15. The first one sets the program bank that is valid for everything, except for these *two* opcodes: *LDA ($xx),Y* and *STA ($xx),Y* (the source or destination bank of which is specified by the other).

Actually there are only fifteen banks, since the last one is the system bank. Similarly to the C128, but even worse again, as it neither has a single byte of memory being shared with any of the other banks… And that is why this architecture is referred to as a “bank switching nightmare”. The programming is complicated, clumsy. The system bank only has a few kilobytes of RAM (besides ROM and I/O), of which only 1K is free and usable (at $0400). So you can either run your machine language code from there, or in any of the separated other banks: with lots of free spaces, but also with losing connection with almost the whole system (including the Kernal calls and interrupts). Because of these circumstances, the CBM-II stays always an odd platform: a strange beast… that requires a raw deal (a special code).

While programming in Basic, it does not seem so tragic yet. The machines have 128K or 256K memory by default, one bank for the Basic program code, and the other(s) for the variables can be used. So the Basic area starts at $0003 here (and the pointers are at $2d-$2e like on the C128/65). The screen is mapped to $d000 in the system bank (either 2K for 80x25 in monochrome, or 1K + 1K colour RAM at $d400 for 40x25 on having a VIC-II instead).

There are also BANK commands for targetting for PEEK and POKE (similarly to the C128), but the applied bank numbers are slightly different: on the *5x0* line (P-series) BANK 0 is the Basic program bank, and BANK 1 is the variable bank; while on the *6x0* line (B-series) the BANK 0 is not in use, and BANK 1 is the program bank, leaving three (BANK 2-3-4) being wasted for the variable area. In Assembly, the equivalent of this is to simply write the bank number into $01 (as the above-mentioned two opcodes are used for PEEK and POKE).

The last two bytes of Kernal are $fbe5 (on the 5x0) and $fbd6 (on the 6x0).

Summary and Comparison Tables

Computer system identification:

peek(65535) =	228/230	->	PET	(rev.4/rev.1-3)
	250/251	->	C65	(Rev2B/other)
	250	->	LCD	
	251	->	CBM-II	
	255	->	(other)	

peek(65534) =	72	->	C64	($ff48)
	23	->	C128	($ff17)
	114	->	VIC-20	($ff72)
	14	->	LCD	($fa0e)
	n/a	->	TED (Plus/4)	($fcb3)
	107	->	PET (rev.1-2)	($e66b)
	27	->	PET (rev.3)	($e61b)
	66	->	PET (rev.4)	($e442)
	229	->	CBM-II (5x0)	($fbe5)
	214	->	CBM-II (6x0)	($fbd6)

Start of BASIC memory:

C64	2049	($0801)
C65	8193	($2001)
C128 (char/gfx)	7169/16385	($1c01)
Plus/4 (char/gfx)	4097/16385	($1001)
LCD (char/gfx)	4097/12289	
VIC-20 (stock/8kB/>8kB)	4097/1025/4609	
PET series	1025	
CBM-II (5x0/6x0)	3 (in memory bank 0/1)	

Screen RAM:

C64/128	1024	($0400)
C65/LCD	2048	($0800)
Plus/4 (TED series)	3072	($0c00)
VIC-20 (>8kB/<=8kB)	4096/7680	($1000/$1e00)
PET series	32768	($8000)
CBM-II (5x0/6x0)	53248	($d000) in bank 15

Colour RAM:

C64/128/65	55296	($d800)
C65 (80-column mode)	63488	($f800) in bank 1
Plus/4 (TED series)	2048	($0800)
VIC-20 (>8kB/<=8kB)	37888/38400	($9400/$9600)
CBM-II (5x0)	54272	($d400) in bank 15

Maximal loadable file size:

C64	51 kB	($0400-$cfff)
C65	<60 kB	($1300-$feff)
C128	<60 kB	($1300-$feff)
Plus/4	<62 kB	($0800-$fcff)
C232	<30 kB	($0800-$7ff5)
C16/116	<14 kB	($0800-$3ff5)
PET/VIC-20	31 kB	($0400-$7fff)
CBM-II	<64 kB	($0003-$ffff)
C64 (with $c000 used)	47 kB	($0400-$bfff)
C65 (with $d000 used)	47 kB	($1300-$cfff)
(after all)	<44 kB	($1300-$bfff)
	<28 kB	($1300-$7fff)

So What Is the Least or Greatest Common Set?

As for program space: as can be seen above, if you lean on the full-expanded machines, you may count on some **30-40 kilobytes** of common space (or even more). Which is not so bad… However, if you *really* want to support *all* platforms, then it drastically shrinks, finally quite only to an innermost single 4K block (from $1000 to $1fff). Below that narrow slice some computers have no memory at all (the stock VIC-20) as neither have the others that much free space (since it is used for screen data or other things). Over $2000 some systems have nothing again (the 8K PET/VIC-20) and the C65 also has some problems there (as it is already covered by the system ROM and bank switching is not so easy in this case).

Even worse, on VIC-20 the screen is also there (either at the beginning or the end). The C128 and the C65 systems also use the first three pages there ($10xx holds the definitions of the function keys and $11xx/$12xx some other things). So thus you can only start your program exactly at **$1301** (or $1401) and have only little more than **3 kilobytes**. (And it is even not a 100% success then, because there still remain two exceptions: the 4K PET and the CBM-II series, for whom we should need to find some other solution.)

As for variable space: there are many locations on the **zero page** that can be used. The first few ones are surely free on all machines (from **$02** to **$0f**) and usually the last ones, too (from **$fa** to **$ff**). Also many in-between (but you must be hunting for them one by one).

If you ever need more space, then bravely also think about the **stack**: the very most of which gets never used at all. As a matter of fact, only the uppermost one or two dozen bytes are frequently in use, and sometimes the bottom ones as well. So, say nearly from **$0110** to **$01c0** you may find a safe and solid contiguous block that never really should be overwritten by any of the operating systems on all machines.

As for what to do: firstly try the jumping table of the common Kernal calls – because they had exactly been created for this purpose (e.g. JSR $FFD2 to print a character).

And remember: this is just the *very least* (or in other words the bare minimum). You may have got infinite possibilities, albeit the more you want, the more (and more and more!) difficult and complicated your job will become. (And this is exactly what I am planning to do with Rosetta: a multi-player, MUD-like, real-time text IF to play on all CBM machines.)

*In the following sections, I will share and discuss some programming techniques and ideas. The examples are mostly taken from my **MemTest64** and **SDOS** projects: both of them are Public Domain, open-source and freeware. (It is strongly recommended to download them and read along all the source files and the descriptions that they contain.)*

Let Us Start at $1301 (the Magical Address)

Why exactly there? On one hand, just read it above (we cannot go below); on the other hand, we also need to stay compatible to the Basic conventions. The Basic area must always be preceeded by a single (otherwise not used) zero byte: hence the $xx01 ending everywhere. The Basic starting might be set to this position on each machine. For example in this way:

POKE 44,19 : POKE 4864,0 : NEW *(or using 46 instead of 44 if needed)*

A program can be basically loaded in two ways: either as an ML (i.e. machine language) code (by typing LOAD “filename” ,8,1 and an appropriate SYS command, like SYS 4865), or as a Basic code (by typing LOAD “filename” ,8 and RUN). Also there are further possibilities (e.g. DLOAD, BLOAD, RUN “filename”, BOOT “filename”, autoboot or autostart). And we need to try to satisfy as many of them as possible, in order to be perfect.

That means we need a little Basic code to begin with; which is also necessary by the way, and even worse, this code portion must be prepared to be launched from all kinds of the different locations being used by all machines as Basic start – then be able to *relocate itself* to where we want it to *really* run. This is how my MemTest64 program starts:

```
*=$1301

;       20 i=peek(65534)+peek(65535)*256:a=peek(45)+peek(46)*256:
;          ifpeek(43)=1andpeek(45)=203thena=peek(44)*256+1
;       20 ifi=64014thena=peek(102)*256+1
;       20 b=i:s=a+3211:ifa=3thens=272:fori=sto315:bank-(b=64470):
;          a=peek(i+5452):bank15:pokei,a:next
;       20 ifa>4865thens=272:bank.:fori=sto402:pokei,peek(a+i+3068):
;          next:bank128+113*(b=65303)
;       20 syss

byte   $4c,$0d,$14,$00,$49,$b2,$c2,$28,$36,$35,$35,$33,$34,$29,$aa,$c2
byte   $28,$36,$35,$35,$33,$35,$29,$ac,$32,$35,$36,$3a,$41,$b2,$c2,$28
byte   $34,$35,$29,$aa,$c2,$28,$34,$36,$29,$ac,$32,$35,$36,$3a,$8b,$c2
byte   $28,$34,$33,$29,$b2,$31,$af,$c2,$28,$34,$35,$29,$b2,$32,$30,$33
byte   $a7,$41,$b2,$c2,$28,$34,$34,$29,$ac,$32,$35,$36,$aa,$31,$00

byte   $6c,$13,$14,$00,$8b,$49,$b2,$36,$34,$30,$31,$34,$a7,$41,$b2,$c2
byte   $28,$31,$30,$32,$29,$ac,$32,$35,$36,$aa,$31,$00

byte   $b4,$13,$14,$00,$42,$b2,$49,$3a,$53,$b2,$41,$aa,$33,$32,$31,$31
byte   $3a,$8b,$41,$b2,$33,$a7,$53,$b2,$32,$37,$32,$3a,$81,$49,$b2,$53
byte   $a4,$33,$31,$35,$3a,$dc,$ab,$28,$42,$b2,$36,$34,$34,$37,$30,$29
byte   $3a,$41,$b2,$c2,$28,$49,$aa,$35,$34,$35,$32,$29,$3a,$dc,$31,$35
byte   $3a,$97,$49,$2c,$41,$3a,$82,$00

byte   $f8,$13,$14,$00,$8b,$41,$b1,$34,$38,$36,$35,$a7,$53,$b2,$32,$37
byte   $32,$3a,$fe,$02,$2e,$3a,$81,$49,$b2,$53,$a4,$34,$30,$32,$3a,$97
byte   $49,$2c,$c2,$28,$41,$aa,$49,$aa,$33,$30,$36,$38,$29,$3a,$82,$3a
byte   $fe,$02,$31,$32,$38,$aa,$31,$31,$33,$ac,$28,$42,$b2,$36,$35,$33
byte   $30,$33,$29,$00

byte   $ff,$13,$14,$00,$9e,$53,$00

byte   $00

*=$1400

;       0 sys5133

byte   $00,$0b,$14,$00,$00,$9e,$35,$31,$33,$33,$00,$00,$00

*=$140d
```

As can be seen and thought, the actual main code will only begin here at $140d that equals to a **SYS 5133** command, when loaded as an ML code. (Yet everything else before this point is also needed for getting here otherwise.) Now let us see the Basic lines explained:

```
20 i=peek(65534)+peek(65535)*256:a=peek(45)+peek(46)*256:
   ifpeek(43)=1andpeek(45)=203thena=peek(44)*256+1
```

In the first Basic line, we read our system identifier. (The last two bytes of Kernal at $fffe-$ffff that is after all the IRQ vector of the CPU, and as we have seen before, fortunately unique for each system.) Then we decide which pair of the Basic pointers we need. (On those machines using 43-44, the low byte of them is normally 1, and then the other pair of pointers at 45-46 hold the ending of the program: of which, we also test the low byte, and that happens to be 203 here just because my MemTest64 ends there; so, once that end changes, then it must be altered here, too. On the other machines, the 43-44 pair is meaningless, and usually zero.)

```
20 ifi=64014thena=peek(102)*256+1
```

In the second line, we also check it for the LCD (note that 64014 = $fa0e).

```
20 b=i:s=a+3211:ifa=3thens=272:fori=sto315:bank-(b=64470):
   a=peek(i+5452):bank15:pokei,a:next
```

The third line is mostly for the CBM-II. Note that 272 = $0110, so we shall use the stack here (as a temporary program space). The 3 value in the IF condition identifies that system (which is the Basic start, and not possible otherwise). The BANK switching is also tricky: 64470 = $fbd6 (for the 6x0), so as the embedded logical condition (b = 64470) is evaluated, it gives back -1 when true, and 0 when false: by multiplying it with another -1, we get the correct bank number of BANK 0 or 1 (where the Basic program is).

```
20 ifa>4865thens=272:bank.:fori=sto402:pokei,peek(a+i+3068):
   next:bank128+113*(b=65303)
```

In the fourth line, we do the same for the C128 and the C65 (similarly choosing between BANK 15 and BANK 128). Please note that the Basic dialects are not compatible here: while that BANK statement on CBM-II has a one-byte token ($dc), this “other” BANK statement on C128/C65 already has a two-byte length ($fe, $02).

```
20 syss
```

Finally the fifth line jumps onto the ML code…

…yet there are *three* variations *where*: normally to the Basic start + 3211 (if we assume the Basic start as $1301 = 4865, then it would be $1f8c = 8076), or otherwise to 272 in the stack (where we have copied one of those two temporary codes right before).

In the next sections, we inspect the three temporary ML codes. But before that, please recognize yet another beauty of my Basic gem: it is right exactly round 256 bytes (and it took so many iterations of bit hunting and optimizing until I managed to shrink that to this size). Moreover, still staying within that, the FOR-NEXT cycles are also optimized to be as fast and short as possible (even including to carefully choose the order of the variable declarations, since the earlier ones can a little bit quicker be accessed by the interpreter).

Oh, and yet another final note… the very first three bytes: $4c, $0d, $14. Which happens to be a JMP $140D command (SYS 5133) there, "hidden" in the Basic line (and yes, it is the cause of that strange line number of 20). It was only needed for sake of supporting the BOOT command. (On some machines, if you type in **BOOT "filename"**, it loads the file as an ML code, and instantly starts that, too, by jumping to the first byte of the code.)

Upward Relocation (on C64, VIC-20, TED, LCD and PET)

This is called "upward" because the loaded Basic program resides below the target space (and so will be moved upwards in the memory). In such cases, if there may be any overlap possible between the source and destination areas, the copying of bytes has to be made in decreasing order (from up to down). (If we did it in the opposite direction, then it would overwrite itself somewhere, and so thus would get corrupted.)

The following code part does this job, but first it repeats the same indentifications as the Basic lines before, and calculates the relative address, according to the Basic start. The cycle used for the relocation itself is also first copied into the stack and run from there (at $0110, the *flp2* cycle). It is necessary because if we ran it from within the same space being copied momentarily, then it would overwrite itself, too (and crash).

Downward Relocation (on C128 and C65)

This part is already being run within the stack (so it needs not to be copied there). The main code will be moved downwards, so thus it must be made in increasing order now. And there is yet another difficulty: because it may (at least partly or even entirely) be covered by the Basic or the Kernal ROM, we first need to page the ROM out (and after the copying page back). On the C128, it occurs from $4000 onwards, and the solution is easy: just write $3f to $ff00 for paging it out (and then $00 for paging it back). On the C65, however, it is not so easy: we must use the MAP command of the processor (a special extension of the 4510 CPU), and it also occurs already from $2000 onwards then. (So normally the whole Basic program area is covered, and this is why it cannot be launched by using just a single SYS command.)

*Here follow both two parts together (once at **$1f8c** and then again at **$200d**, also with another Basic line inserted in-between which might be used if loaded with ,8,1 on C65):*

```
*=$1f8c

;       upward relocation (sys8076) <- Basic + 3211

        sei
        ldx     $2b
        lda     $2d
        ldy     $2e

        cmp     #<ende
        bne     bex
        dex
        bne     bex
```

```
        lda     #$01
        ldy     $2c

bex     ldx     $fffe
        cpx     #$0e
        bne     fex
        ldx     $ffff
        cpx     #$fa
        bne     fex

        lda     #$01
        ldy     $66

fex     sec
        pha
        sbc     #$01
        pha
        tya
        sbc     #$13
        tax
        pla
        php

        adc     #<flp2-1
        sta     $02
        txa
        adc     #>flp2
        sta     $03

        plp
        pla
        bcs     und

        adc     #$ff
        sta     $04
        tya
        ldy     #$1a
        adc     #$15
        sta     $05

flp1    lda     ($02),y
        sta     $010f,y
        dey
        bne     flp1

        lda     #$29
        ldx     #$16
        sty     $02
        sta     $03
```

```
        jmp     $0110

flp2    dey
        lda     ($04),y
        sta     ($02),y
        tya
        bne     flp2

        dec     $03
        dec     $05

        txa
        dex
        bne     flp2

und     cmp     #$01
        bne     down
        txa
        bne     down

        jmp     main

*=$2000

;       0 sys5133

        byte    $00,$0b,$20,$00,$00,$9e,$35,$31,$33,$33,$00,$00,$00

*=$200d

;       downward relocation (sys8205) <- Basic + 3340
;       (copied to $0110 and started there)

down    sei
        ldy     $2b
        lda     $2d
        ldx     $2e

        dey
        bne     xex
        cmp     #<ende
        bne     xex

        ldx     $2c
        bcs     smex

xex     ldy     $ffff
        cpy     #$fa
```

```
        bne     lex
        ldy     $fffe
        cpy     #$0e
        bne     kex

        ldx     $66
        ldy     #$00

smex    inx
        sty     $02
        stx     $03

        php
        bcs     zyex

kex     ldy     $ffff

lex     clc
        adc     #$ff
        sta     $02
        txa
        adc     #$00
        iny
        sta     $03

        php
        beq     qyex

        lda     #$00
        tax
        tay

;       TAZ / MAP

        byte    $4b,$5c

        lda     #$07
        ora     $00
        sta     $00

        lda     #$f8
        and     $01
        sta     $01

zyex    lda     #$14
        ldx     #$16
        sty     $04
        sta     $05
```

```
flp3    lda     ($02),y
        sta     ($04),y
        iny
        bne     flp3

        inc     $03
        inc     $05

        dex
        bne     flp3

        plp
        bcs     sjex
        beq     stex

        lda     #$07
        ora     $01
        sta     $01

        tya

;       LDX     #$E3
;       LDZ     #$B3

        byte    $a2,$e3
        byte    $a3,$b3

;       MAP / TAZ

        byte    $5c,$4b

sjex    jmp     main

qyex    ldx     #$3f

stex    stx     $ff00
        bne     zyex
        beq     sjex
```

A little more about the C65 mapping: on this machine, the bank switching is done via its integrated mapper function, being invocated by the **MAP** command ($5c opcode). That needs all four registers being set before: one pair for the lower and the other pair for the upper 32K (so both halves can be mapped to any extended memory location independently of each other; whether it be RAM or ROM). This method is rather flexible, albeit a bit complicated (and we must also be sure if the 4510 instruction set is at present). E.g. for BANK 0 (which is the same layout as that of the C64 mode by the way) all four registers need to be set to zero. Once a MAP is used, at any time later an **EOM** ($ea opcode, identical to that of the NOP) must also be applied (or else the interrupts will not work because of staying hung up for ever).

And a Third One for the Joker (on CBM-II)

This third temporary code part (already copied to and running in the stack) copies its own main code (apart from the others) from the Basic bank (0 or 1) into the system bank (15) at $0400 (up to 1K of size at maximum) and jumps to there. (In Rosetta, I will rather make it in the other way around: I am running the common main code in the Basic bank. But that requires a very complicated bank switching environment implemented for the interrupts and Kernal calls… So we do not do so here.)

```
*=$295a

;       CBM-II relocation (10586) <- Basic + 5721
;       (copied to $0110 in system bank and started there)

        sei
        lda     $fffe
        ldx     #$01
        cmp     #$d6
        beq     cbm2
        dex

cbm2    ldy     #<$1688
        lda     #>$1688

        stx     $01
        sty     $02
        sta     $03

cbmy    ldy     #$00
        ldx     #$03

cblp    lda     ($02),y
        sta     $0400,y
        iny
        bne     cblp

        inc     $03
        inc     $012d

        dex
        bne     cblp

        jmp     $0403

*=$2986

;       (copied to $0400 in system bank and started there)
```

Another Relocating Method (Not Basic Dependent)

The above-mentioned method with the Basic header and the corresponding temporary codes is well capable to launch your application on each system and (almost) from any Basic start address. But it has a weakness, too: it does need that Basic start. There might be a situation when you want to write a pure Assembly code which is similarly able to relocate itself. Here follows a theory on how to do it then. So the program has just begun, and it needs to find out where it is – but it cannot use yet any fix addresses until we do the relocation.

We shall use the PC (Program Counter) register of the CPU; however, it is not directly accessible unfortunately. Still accessible indirectly: for example through a dummy call:

```
sei
lda     #$60
sta     $02
jsr     $0002
```

What happens here? We place an RTS opcode at $0002, and then call it, that immediately returns from the “subroutine”. What is interesting now, that is the side effect: the returning address still remains on top of the stack. (The SEI is needed to avoid that from being accidentally overwritten by an interrupt.) The following part reads it from there:

```
tsx
sec
dex
lda     $0100,x
sbc     #<addr
sta     $02
lda     $0101,x
sbc     #>addr
sta     $03
```

And it also compares it (by making a 16-bit comparison via two subtractions) with another address (which we must already calculate before, at coding time, and write its low and high bytes in place of those “addr’s” above). We need to know that the address found on the stack will be *one byte less* than the real returning address: because all the 65xx CPU’s are working in this way. So it should be at the third byte of the JSR $0002 command exactly (that is $20, $02, $00 in the memory in real, and it should point to that last zero byte here). Thus, the absolute address of this position (in our actual code compiled) must be used as “addr”.

In $02/$03, we have now got the *difference* (i.e. how many bytes away the relocation goes), and we also have a sign in the C flag. If C=0, we are at a lower address momentarily, so we need an upward relocation in this case. If C=1, we are at higher, so a downward relocation must be done (or the addresses could even be equal, and then no relocation is needed):

```
bcc     upward
ora     $02
beq     ready
bcs     downward
```

If we now add the absolute destination address of the main code (i.e. that part to be relocated and then be jumped onto later) to this difference, then we get the source address in $02/$03 (and the destination address in $04/$05 as well) for the relocating cycle:

```
clc
lda   #<main
sta   $04
adc   $02
sta   $02
lda   #>main
sta   $05
adc   $03
sta   $03
```

This can be applied for the downward relocation (which is made in increasing order as we know). For the upward relocation, however, we rather need to calculate the ending of the main code part instead (since it is made in decreasing order).

(If you want to see this method “in action”, you may find it in the SDOS, and later also in the Rosetta source code, when published.)

A Platform Independent Autostart

Autostart is generally meaning that you need no SYS nor RUN command to be typed in; but rather have another very little extra file (usually only one or two blocks of size on disk) which starts automatically right after having been loaded, and then also boots the main file in. There are many different well-known methods on the C64 to achieve this, but I have only found one of them, that can be also used as a platform independent one: the *stack* one.

It relies on the circumstance that the stack is always in use, also during the LOAD routine, and it holds the returning address from there: although we do not exactly know where, but we do not need to know it, either, if we fill it with the same value (so the low and high bytes of the address are the same), thus redirecting it onto a fix, known address (where we place a special code be prepared to continue). There are basically two choices possible: either use the $01 or the $02 value everywhere. As we know it, the address found on the stack will be always one byte less than the real returning address (i.e. it is still incremented by one on RTS). So it means that we can do the redirection either to $0102 or $0203.

In the first case, we can make an exactly *one-block* boot loader file on disk (which is 252 bytes of size): at $0102-$01fd (which also means we lose control over the last two bytes of stack, but it is no problem as they do not really matter here).

In the second case, a *two-block* sized boot loader is possible: at $0100-$02a8 (since $02a9 is already used by the operating system, so it is the maximum).

The first case has a bit smaller size, yet better compatibility: it must work almost on all machines. The second case does not always work on all machines. It has two main causes: on one hand, the second block is loaded *later* than the stack is filled (so if it gets already fired

by an RTS, while the corresponding code is not loaded yet, then it may crash), and on the other hand, that area over $0200 may be used by some of the operating systems, too.

In both cases, the ending of the stack must be filled with the same bytes (either $01's or $02's) at least from $01e0 (or earlier) until $01fd (or $01ff). And we must prepare a code part waiting to be fired off at $0102 or $0203 (when the LOAD routine gets to an RTS, it indirectly jumps to there, instead of the normal returning).

Both SDOS and Rosetta make extensive use of this method (especially for the latter, you may find an entire series of such boot loaders, one for each system the best optimized).

Boot or Autoboot

Booting is possible on C128 and C65 systems (including the MEGA65). Yet the actual process is quite different on them. What the same is: it is executed either by manually typing in the BOOT command (without any specified filename), or by putting in the boot-capable medium (normally a floppy disk or SD card) and turning on or resetting the computer.

On C128: the operating system looks for the first block, or called as the boot block (which is on the track 1 and sector 0) on the disk, and if it has some special content, then loads it at $0b00 and executes it from there.

It is also emulated by the SD2IEC drives (even directly from the SD card without mounting a disk image) where a **BOOTSECT.128** file must be placed into the root directory of the SD card (and it must be set to 8 as drive number).

It may either contain a Basic line to be executed (which may load and run an application, or do any other things), or a little ML code. The official format of preparing a boot sector must look like the following (however you might also use up that whole space for your own program code at your will as well, of course):

```
*=$0b00

;       "cbm" (autoboot signature)

        byte    $43,$42,$4d

;       load address for additional sectors (T1, S1)

        word    $0000

;       bank number for additional sectors

        byte    $00

;       number of sectors to load

        byte    $00
```

```
;       boot message: "booting…"
;       (or a single zero byte for no boot message)

        byte    $42,$4f,$4f,$54,$49,$4e,$47,$2e,$2e,$2e,$00

;       program name to load on boot (a filename and/or just a zero byte)

        byte    $00

;       to execute a Basic command (as a string)

        ldx     #<cmd-1
        ldy     #>cmd
        jmp     $afa5

;       (here is the string)

cmd     byte    $00

;       (finally filled with zeroes up to 256 bytes of size)

*=$0c00
```

On C65: it is much simpler there. The drive 8 will be searched by the operating system for a specific filename **AUTOBOOT.C65***, and if found, then it will be loaded and run as a normal Basic program file.

Detecting If PAL or NTSC System

Some of the systems do not depend on the TV regulation as they have no TV output (PET, CBM-II and LCD). Some of them have two different Kernals hard-coded (VIC-20 and the TED series). And the remaining three (C64/128/65) have the same Kernal for both standards and make an autodetection at booting time to specify which one they are actually running on. You may read the result of that from a given space (at $02a6 on C64 where 0 means the NTSC and 1 signs the PAL; or at $0a03/$1103 on C128/65 where 0 or $ff stands for the NTSC or the PAL). The problem with it is that these values are not reliable: the autodetection is faulty (so thus it is quite "normal" on C64's with CPU accelerators like the SuperCPU or Chameleon, that a PAL machine is detected as NTSC by mistake), and even if it works well, the result in RAM might be overwritten by another application or the user.

This is why you cannot lean on them, but rather make your own detection, by inspecting the physical hardware itself. That is quite easy as a matter of fact: you must only count the scanlines on the screen (by looking for a highermost value different from $ff in the raster counter). Normally a PAL screen has 312 scanlines, while an NTSC one has about 263 lines (or plus/minus one or two sometimes).

The following code part does this job for you (and gives back 0 or 1 in the X register for the NTSC or the PAL):

```
        sei
        ldx     #$00

ntpx    txa

ntpl    tay
        sta     $02
ntpr    lda     $d012
        cmp     $02
        bcs     ntpl

        iny
        beq     ntpx

        inc     $02
ntpa    bit     $02
        lda     $02
        cmp     #$10
        bcc     ntsc

pal     inx
ntsc    cli
```

This works for the C64/128/65 in this way, and it can be made working for the VIC-20 and the TED's, too, by some self-modification:

On TED machines: only the $d012 value at "ntpr" must be changed for $ff1d.

On VIC-20: that same value must be changed for $9004 instead, and also the BIT opcode at "ntpa" must be changed for ASL (by writing $06 to "ntpa"). This latter modification is needed because the VIC-I chip only counts every second scanline in the raster counter (so it must be multiplied by two).

Finally, two additional notes on this:

Note 1: on C128, the VDC chip operates totally independently of the VIC-IIe timing: the VDC itself is the very same on both PAL and NTSC motherboards, and its timing will be only programmed by the operating system (to be the same as that of the VIC-IIe chip). You may even manually re-program it to have a different output (by specifying the screen size in some corresponding registers) if you want. (Even for a direct VGA or anything!)

Note 2: on C64, it is often rather difficult (or even close to impossible) to make an NTSC version of a PAL game (or a demo effect), since those machines have considerably less computing time per frame (because of using 60 frames instead of 50 per second). If you are working on such a game or demo, and you do not want to sacrifice any of your features (nor simplify it in any other way) in order to make it NTSC compatible, just apply the well-known C128 trick of 30% CPU gain in C64 mode instead (by switching on and off the $d030 fast mode bit on raster interrupt). It fixes that for Turbo Chameleon (and MEGA65 and TDC), too.

Measuring the MHz of CPU

Here follows a method (both used in MemTest64 and Rosetta) which is suitable for measuring the actually perceived speed of CPU. It is pretty exact on 65xx CPU's, at least up to two decimal places (which is more than enough for any practical purpose). For example, the 6510 CPU of C64 is measured with screen enabled as 0.93 MHz on PAL and 0.96 MHz on NTSC machines; and with screen disabled as 0.98 MHz on PAL and 1.03 MHz on NTSC machines. (While the 8502 CPU of C128 in fast mode is measured as 1.88 MHz on PAL and 1.97 MHz on NTSC machines.)

The main principle is quite easy: exactly for 0.1 second, we are counting the executed cycles by the CPU. Whereas 100.000 cycles for this time period signs a "hypotetically" exact 1 MHz clock. If we measure more or less, the clock is also proportionately more or less.

To do it well, we need to rely on the result of the previous section code (*Detecting If PAL or NTSC System*, see before), and it is also strongly recommended to execute this code part right after the above. So it starts already with X register set to 0 or 1 (for NTSC or PAL):

```
        sei
        lda     #$00
        sta     $02
        sta     $03

        txa
        clc
        eor     #$01
        adc     #$05

;       0.1 sec = 5 x PAL or 6 x NTSC frames

        sta     $04

;       raster synchronization

        jsr     waitmp

;       The inner loop takes 99 cycles (from x00 to x09), so repeatedly
;       (2 + 99 x 10 - 2 + 9) x 100 = 99.900 cycles (from xxx to x99),
;       plus few (for a few times when a bit more because of branching)
;       that means just some hardly less than 100.000 cycles after all:

;       ...at least on a 65xx CPU (probably on C65 slightly less),
;       ...and with screen/sprites/interrupts completely turned off.

;       (This program takes only care about disabling the interrupts,
;       thus other things, like the screen and sprites, as well as the
;       fast mode and the turbo settings yet are waiting for the user!)
```

```
yyy     ldy     #$00

xxx     ldx     #$00

x00     lda     $02
        cmp     ($03,x)
        lda     $03
        cmp     ($02,x)

x01     lda     $0100
        cmp     ($04,x)
x02     lda     $0201
        cmp     ($03,x)
x03     lda     $0302
        cmp     ($02,x)

        inc     x01+2
        dec     x02+2
        inc     x03+2

        nop
        nop
        nop
        nop
        nop

rllb    lda     $d012
        and     #$f0
        beq     xiux
        nop
        sta     xiux+1

xuix    inx
        cpx     #$0a
        bcs     x09
        jmp     x00

x09     iny
        cpy     #$64
        bcs     x99
        jmp     xxx

x99     inc     $02
        bne     yyy
        inc     $03
        bcs     yyy

xiux    cmp     #$00
        beq     xuix
```

```
        sta     xiux+1

rlhb    lda     $d011
rlhc    and     #$80
rlhd    beq     xuix

        dec     $04
        bne     xuix

        cpx     #$05
        ldx     $02
        lda     $03
        bcc     roua
        iny
        cpy     #$64
        bcc     roua
        ldy     #$00
        inx
        bne     roua
        adc     #$00

roua    stx     $02
        sta     $03
        sty     $04
        cli
```

It gives back the integer part of the MHz value in $02/$03 (as well as the X/A registers), and the fraction in $04 (as well as the Y register) from 0 to 99 (from $00 to $63). The integer part is a 16-bit number, because it is theoretically possible to be greater than 255 MHz (although there has been no such precedent known yet).

An additional code part:

```
;       raster synchronization

waitmpjsr       wmpp
        bne     waitmp

wmp     jsr     wmpp
        beq     wmp
        rts

wmpp    lda     $d011
wmpa    and     #$80
        rts
```

This works for the C64/128/65 in this way, and it can be made working for the VIC-20 and the TED's, too, by some self-modification:

On TED machines: the $d012 value (at “rllb”) must be changed for $ff1d; the $d011 values (at “rlhb” and at “wmpp”) must be changed for $ff1c; and the $80 values (at “rlhc” and at “wmpa”) must be changed for $01.

On VIC-20: the $d012 value (at “rllb”) must be changed for $9004; the BEQ opcode at “rlhd” must be changed for AND ($29); and the JSR opcode at “waitmp” must be changed for RTS ($60).

When compiling the above codes (and especially the inner loop), please pay attention to their alignment in memory (e.g. not to have a page boundary anywhere within), since a bad alignment might cause some false results (by altering the number of the executed cycles).

Printing a 16-bit or 24-bit Integer on All Machines

```
;       printing 16-bit/24-bit number in $02/$03/$04

        ldx     #$00

num16   stx     $04
num24   stx     $05
        ldx     #$08
        bne     nlp0

nlp1    lda     $02
        sbc     ntab1-1,x
        sta     $02
        lda     $03
        sbc     ntab2-1,x
        sta     $03
        lda     $04
        sbc     ntab3-1,x
        sta     $04
        iny

nlp2    lda     $02
        cmp     ntab1-1,x
        lda     $03
        sbc     ntab2-1,x
        lda     $04
        sbc     ntab3-1,x
        bcs     nlp1
        tya
        bne     nlp3
        ldy     $05
        beq     nlpy

nlp3    ora     #$30
        sty     $05
        jsr     $ffd2
```

```
nlp0    ldy     #$00
nlpy    dex
        bne     nlp2

        lda     #$30
        ora     $02
        jmp     $ffd2

ntab1   byte    $0a,$64,$e8,$10,$a0,$40,$80
ntab2   byte    $00,$00,$03,$27,$86,$42,$96
ntab3   byte    $00,$00,$00,$00,$01,$0f,$98
```

Detecting the Instruction Set of CPU

Also quite easy: just execute a few (but carefully chosen) opcodes that make different things on different CPU's – and compare the results. *There are basically four cases:*

0: 65xx (including 6502, 6509, 6510, 7501, 850x and 65DTV02 etc.)
1: 65CE02 (or 4510 in the C65 and MEGA65)
2: 65C02 (or 65C102 in the LCD)
3: 65816 (on SuperCPU and other accelerator cards)

The above value (0-3) will be given back by the next code part:

```
        sei
        lda     #$00

;       $1A = INC A on newer CPU but only a NOP on 65xx

        byte    $1a,$ea,$ea
        beq     noxp

;       on 65CE02: ROW $EAA9 (a = $01)
;       on 65C02: NOP / LDA #$EA (a = $ea)
;       on 65816: XBA / LDA #$EA (a = $ea, b = $01)
;       (on 65xx: SBC #$A9 / NOP)

byte    $eb,$a9,$ea

        cmp     #$01
        beq     nocp

;       (on 65CE02: LDA #$00 / ROW $1A1A)
;       on 65C02: LDA #$00 / NOP / INC A / INC A (a = $02)
;       on 65816: LDA #$00 / XBA / INC A / INC A (a = $03, b = $00)
;       (on 65xx: LDA #$00 / SBC #$1A / NOP)

byte    $a9,$00,$eb,$1a,$1a
```

```
        cmp     #$04
        bcc     nocp
noxp    lda     #$00
nocp    cli
```

It includes some surplus redundancies (like those two NOP's following the INC A and so on) that are only left inside in order to stay absolutely safe and sure in all conditions.

A note on MEGA65: this machine is promised to should be able to switch over the instruction set between the 4510 and the 6510 ones (e.g. when going to C64 mode), so the actual result will depend on this setting. ***+note:***(2)***

Fast and Slow (on C65, MEGA65 and DTV)

MemTest64 provides some subroutines for them (can be called in C64 mode):

```
*=$2800

;       MEGA65 = fast mode <- SYS 10240

m65on sei
        lda     #$47
        sta     $d02f
        lda     #$53
        sta     $d02f
        lda     $d054
        ora     #$40
        bne     mfs

*=$2812

;       MEGA65 = slow mode <- SYS 10258

m65off sei
        lda     #$47
        sta     $d02f
        lda     #$53
        sta     $d02f
        lda     $d054
        and     #$bf
mfs     sta     $d054
        lda     #$ff
        sta     $d02f
        cli
        rts

*=$282c
```

```
;       C65 = fast mode <- SYS 10284

c65on   sei
        lda     #$a5
        sta     $d02f
        lda     #$96
        sta     $d02f
        lda     $d031
        ora     #$40
        bne     cfs

*=$283e

;       C65 = slow mode <- SYS 10302

c65off  sei
        lda     #$a5
        sta     $d02f
        lda     #$96
        sta     $d02f
        lda     $d031
        and     #$bf
cfs     sta     $d031
        lda     #$ff
        sta     $d02f
        cli
        rts

*=$2858

;       DTV = fast mode <- SYS 10328

dtvon   sei
        lda     #$01
        sta     $d03f
        byte    $32,$99
        lda     #$03
        byte    $32,$00
        lda     #$20
        bne     dts

*=$2868

;       DTV = slow mode <- SYS 10344

dtvoff  sei
        lda     #$01
        sta     $d03f
        byte    $32,$99
```

```
        lda     #$00
        byte    $32,$00
        lda     #$00
dts     sta     $d03c
        lda     #$00
        sta     $d03f
        cli
        rts

*=$2880
```

A note on MEGA65 (again): the actual switchover between the fast and slow modes on this machine is identical to that of the C65 (as the FAST and SLOW commands in native mode are part of the same Kernal ROM code being used). Therefore the SYS calls should also be the same (changing $d031).

The extra ones for MEGA65 here (changing $d054) are toggling the extra bit used for also switching over between the two kinds of fast modes: either 3.5 MHz (as the same as the normal C65 speed) or 40 MHz or more (the maximum of MEGA65).

You may also use these two shortcuts: POKE 0,65 and POKE 0,64.

A note on DTV: please *only* use these subroutines on DTV; they will crash for sure on stock C64 (the $32 opcode causes a CPU jam there). (The C65/MEGA65 ones are harmless.)

Yet another note: the emulated C65 (in MESS) is measured (by the method that has been published a few sections earlier here) as 4.14 MHz in fast mode, and as 1.18 MHz in slow mode (also in the C64 mode). That is quite normal, as mentioned before: since the CPU is already some 20% faster in advance (because of the faster timing as well as the absence of the VIC-II badlines), and it also matters here. ***+note:***(2)***

The emulated DTV (in VICE) is measured as 1.75 MHz on PAL and 1.82 MHz on NTSC in fast mode; the emulated SuperCPU as 8.83 MHz on PAL and 9.22 MHz on NTSC.

(There are no data at present on the MEGA65 yet, when writing this, nor about the original real hardware.) ***+note:***(3)***

Fast and Slow (on TED)

Disabling the screen on C64 gives only about +5% gain (0.98 vs 0.93 MHz), whereas the same on the TED series is already near to +50% (1.70 vs 1.14 MHz) approximately. Which is thus strong enough to call it as a “fast mode”. You may achieve it in this way:

```
        lda     #$ef
        and     $ff06
        sta     $ff06
```

And then back into the “normal mode” (enabling the screen):

```
lda     #$10
ora     $ff06
sta     $ff06
```

(Simply at the Basic command prompt just use POKE 65286,11 and POKE 65286,27.)

Other than that, they also have a specific “slow mode” (that is slower than the normal) at about 0.78 or 0.89 MHz (on screen enabled or disabled), which might be achieved this way:

```
lda     #$02
ora     $ff13
sta     $ff13
```

As a matter of fact, the normal operating mode consists of continually oscillating between the slow and fast modes (which is done automatically by the TED hardware), and the above commands disable this function. (This is the equivalent to that of the C64’s VIC-II chip “stealing away” some cycles from the CPU in badlines, however it is valid for the entire visible screen area here.) An unpleasant side effect of this is that in normal mode you can never know exactly on which speed the CPU at the moment is (as it is always being altered in the background), that makes it harder to write a cycle-exact timing hungry code (like e.g. a fast loader). So you better choose one of the fix (either fast or slow) speeds then. (And this is why, for example, the official TED version of JiffyDOS disables the screen, whereas it is not so necessary on the other CBM machines.)

Furthermore, on top of that “fast mode” (aka screen disabling or blanking), these computers also have a so-called (unofficial, overclocking) “turbo mode”: at least on the PAL machines, where it means to turn over the TED video chip to NTSC mode, this way:

```
lda     #$40
ora     $ff07
sta     $ff07
```

On PAL machines, it has the side effect (except the display seeming rambled, and so needing to be blanked before) of CPU speed increasing near some 2 MHz or so (that is even faster than the fast mode of the C128!). Unfortunately, on NTSC it (i.e. turning to PAL mode) is rather counterproductive, as it works the other way around: decreasing to about 1.4 MHz. *(My method is not suitable to see these given values as its reference of raster lines is also affected, so the results will be invalid. Rather use a benchmark test for this purpose.)*

The PAL/NTSC models have physically different clock generator chips (of about 17.7 or 14.3 MHz) and their frequencies are divided back (by 20 or 16) to get the 0.89 MHz base clock: only the divisor is swapped on soft-switching, thus 1.11 or 0.72 MHz is got instead.

This soft-switching of PAL/NTSC modes is not emulated by the VICE emulator, so it can be used for emulator detection here: by combining it with measuring the MHz both before and after, and comparing the results. (If it is not changed, then it is an emulation.)

However, it is well emulated by the YAPE emulator (which is a far better one for this purpose, and strongly recommended to be used for TED code instead of the VICE).

Emulator Detection (on C128)

The only emulation of the C128 suitable for normal everyday usage is the VICE emulator these days (and also the C64 Forever, of course, which is also compiled out of the VICE sources). Although the emulation is already quite advanced by now, it still lacks some features of, as well as has some differences compared to the real platform. So the situation is somewhat similar to that of the C65 and the MESS; however, the problems caused by this are minor and fortunately of not so significant importance. The most of them are around the VDC emulation: there are no interlace modes yet (so no IHFLI pictures, Basic 8 or Graphic Booster etc.), and the ready bit of VDC will never get busy.

The latter might be rather *better* eventually this way (as the programs can run faster, smoother without waiting for that bit), yet it generates an indirect and not too pleasant side effect: if anyone develops a software only in the emulator (which depends on this feature even if he does not know of that), then it may happen to *not* run on the real hardware.

There are some similar artifacts around the VIC-IIe emulation, too: the display output does not get garbled when turning on the fast mode (which is also better as a matter of fact this way, yet the above-mentioned side effect is especially not so lucky here), and the test bit does nothing in the emulator. (The test bit is neither considered too useful on the real hardware, although might be used for generating interlace.)

Because of these circumstances, the emulation might even be considered as almost some kind of “new platform” apart from the hardware (at least by considering how many people use it who have no hardware) as well (like we do so in the case of C65, MESS and MEGA65), and at least when writing a new program, *you always have to test it on a real machine*; while it is also highly recommended to test in the emulator, too. Even better, if your program may be able to decide, and choose from both of them: then it may enjoy the advantages of the emulation as well (besides merely avoiding the problems).

For example, if you know it is an emulation, you can use the fast mode all the time.

That is why an emulator detection is a very useful thing. Whether the ready bit or test bit makes it possible; now let me show you my own solution built upon the latter.

The test bit of the VIC-IIe (at $d030/$02) is a strange one: as it is not clear what the goal of the designers was with it. Once it is set, it first speeds up the raster counter, then stops it (the display output thus “flies away”). More exactly: the raster counter starts increasing very quickly (by one per each machine cycle), until it overflows to zero, and then stops: i.e. stays permanently zero (until the bit is cleared again). However, in the emulator, it does nothing at all. So we only need to set it for a while and check the behaviour of the raster counter. (And it has the same effect in C64 mode, too.)

Some careful arrangements must be made before: 1.) the screen must already be blanked out (and the interrupts be disabled), 2.) we must know for sure if it is a C128 (even if in C64 mode), 3.) we better do the other emulator detection at first (by reading from $dfff as mentioned before), because if that one succeeds, we do not need this one.

The detection of C128 (whether in native mode or in C64 mode as well) can be done by writing zeroes to $d02f and $d030, and reading them back: the first one must give back $f8 (as it has three bits for scanning the extended keyboard; or a value between $f8-$fe if such a key is pressed) and the second one $fc (as it has two bits, one for the fast mode, and the other is the test bit); since the unused bits always have a high state (like always being “set”).

There are also further advantages of the emulator: for example, if you are about to write a mouse-driven application, then it can be integrated into the normal Windows environment, similarly to e.g. some such DOS applications that run also emulated in a DOS shell (like the Star Commander or the 64Copy). If the emulator is running in a window (instead of a full screen), then the Windows’ mouse-clicks upon that are translated to emulated lightpen events for the C64 program by VICE: and so thus the Windows’ mouse can be used instead of an emulated Commodore mouse (and you need no sprites on screen).

Or another nice thing is to reach all files on your hard drive from within your C64 program through the integrated virtual file system of VICE as a mass storage (which works more or less similarly like having an emulated 64HDD built in, but also much faster). Or to use the warp mode as a turbo… And so on.

We do not need to fear, either, what happens if the emulators will be further developed in the uncertain future: the worst case scenario is only to “fall back” to the level of the real hardware later, when and if the emulation becomes indistinguishable from that, one day. (Also remember: if your program runs well on real hardware, but not in the emulator, then it is *always* the fault of the emulator, and not yours.) *Here follows my code (as used in Rosetta):*

```
;       (In earlier emulators, e.g. in VICE till v2.2, if the "Emulator
;       Identification" is enabled, $dfff toggles between $55 and $aa
;       whenever is being read. From v2.3 on, it works no more.)

emu     lda     $dfff
        sta     $02
        bpl     ema
        eor     #$ff
ema     cmp     #$55
        bne     xaft
        lda     $dfff
        eor     #$ff
        inx
        beq     kamu
        cmp     $02
        beq     emu

xaft    ldx     #$00
        lda     #$02

;       (In Rosetta, this variable has been used for machine detection
;       before, so we only check it here for having a C128.)

        cmp     $ff
```

```
        bne     faft

        sta     $d030
        lda     $d030
        cmp     #$fe
        beq     xamu
        bne     kamu

;       Detecting emulator by using the test bit:

;       We are following the raster counter, and if it is incrementing
;       normally, then the test bit is not working at all: that must be
;       in VICE. However, if it suddenly jumps forward or stops, that
;       might be a real hardware as well (or a better emulator).

;       (On a real hardware, after the test bit is set, the raster
;       counter will be incremented by one in every cycle until its
;       next overflow: and then afterward it stays forever zero. Only
;       when the test bit is cleared, starts counting again.)

xemu    lda     $d012
        cmp     $02
        beq     xemp
        inc     $02
        cmp     $02
        beq     xamp
        cmp     #$00
        bne     kaxt

xamu    lda     $d012
        sta     $02
xamp    stx     $03

xemp    inx
        bne     xemu

        lda     $03
        bpl     kaft

        ;       (In Rosetta, this variable bit is set to sign the emulator here.)

kamu    tya
        ora     #$10
        sta     $fe

kaxt    ldx     #$00
kaft    stx     $d030
faft
```

Jumping from Native Mode to C64 Mode

The C128, and the C65 (and of course the MEGA65) as well, can execute the GO64 function also through a normal Kernal call, which can be simply called from Assembly (either by JMP or JSR, as there will be no return). That means it is possible to load a C64 application in the native mode, then jump over to C64 mode, and start it there. (It is also possible because the same BANK 0 is used for C64 mode, and its content is not cleared, only a few bytes will be overwritten by the operating system.) But what is the point of that?

First, it is much *faster* in the native mode to load anything (both because of the fast mode of the CPU and the burst mode of the more advanced fast serial IEC protocol).

Second, you may load a *larger* file in native mode. (Remember that it is possible from $1300 to $feff which is 59K, while in C64 mode only from $0800 to $cfff which is 50K.)

And third: it can even be *autobooted* there (just see above for autoboot!). So you need to only turn the computer on, and wait while it does this all for you… like an Amiga.

However, it needs some rather tricky and complicated programming. I will only describe the theory on how to do it now (step by step). But this method should be extensively used by SDOS, and would be a feature of my upcoming ***SDOS 2017*** version ***+note:***(4)***; thus you will be able to find all the corresponding source codes together with the executables "in action" there – coming soon. (It has not been published yet, when writing this, but I make it later in this year.) In theory, the C65 version must work similarly to that of the C128, only with some minor modifications (actually self-modifications made by the code to itself). (Still, in practice, only the C128 version has been tested and implemented momentarily.)

The main principle is that a C64 program normally starts at $0801: this must be first loaded on the "native side" at $1301 instead, and then the code must jump into the "C64 side", and make a downward relocation of the program to the original address ($0801, but it might be any other address as well, of course), and finally start it there (normally with RUN).

Since we want to use the memory over $1301 for the program to be loaded, it means that our code must be somewhere below that. The $10xx page is one of the best places (as it contains the function key definitions, which will be no longer needed for sure, so it can be overwritten, and the system does not touch it otherwise), yet you may also use the most pages between $0bxx and $0fxx, too (on C65 this is the screen!), as well as the stack and zero page.

The GO64 function makes a full *cold reset* on the C64 side: we actually lose control at this point… We can only get control over the system back again by making some dirty hack.

Just place the following nine bytes starting at $8000:

```
word   $1000
word   $fe5e
byte   $c3,$c2,$cd,$38,$30
```

This simulates the presence of a cartridge for the C64 Kernal, which checks for the signature "CBM80" during the cold start (the last five bytes are actually these characters

above), and if found, that makes it jump to the custom cold start vector at $8000-$8001: so thus it will jump to $1000 in this case.

Note: please do not forget to save the original nine bytes out of this space to a safe location before overwriting them, since they are also part of the loaded program, therefore must be restored later on the other side. It is also strongly recommended to save the last used device number (at $ba, the same on all three machines) and later to restore, since it holds the device number from where the program has been loaded, and the program may require that for further disk operations in the future. (It is only reset to zero by the operating system.)

After having been everything prepared, we call the GO64 Kernal function on C128 by:

```
jmp     $ff4d
```

Or the same one on the C65 by:

```
jmp     $ff53
```

Another important difference between the C128 and the C65 versions is the handling of the memory paging – which we also need to do to reach the corresponding RAM on both machines. In order to be able to both read and write those nine bytes above at $8000, we need to page out the system ROM (and afterwards page it back), because that location is always covered by some of the system (either the Kernal or the Basic) ROM's.

On C128, we page the ROM out (which equals to a BANK 0 in Basic) by this:

```
lda     #$3f
sta     $ff00
```

And page it back (aka BANK 15) by this:

```
lda     #$00
sta     $ff00
```

On C65 in native mode, we must use some 4510 opcodes instead. And we also need to set the $00 and $01 ports like we do so in the C64 mode, moreover, to maintain the $d030 register (since it also has some paging bits of ROM's). To page it (all) out (aka BANK 0):

```
sei
lda     #$a5
ldx     #$96
ldy     #$ff

sta     $d02f
stx     $d02f

lda     $d030
sta     rest+1
```

```
        and     #$06
        sta     $d030
        sty     $d02f

        lda     #$07
        ora     $00
        sta     $00

        lda     #$30
        and     $01
        sta     $01

        iny
        tya
        tax
        taz
        map
```

And to page it back (aka BANK 128):

```
        lda     #$00
        ldx     #$e3
        ldz     #$b3
        tay
        map
        taz

        lda     #$07
        ora     $01
        sta     $01

        lda     #$a5
        ldx     #$96
        dey

        sta     $d02f
        stx     $d02f

rest    lda     #$64
        sta     $d030
        sty     $d02f

        eom
        cli
```

Other than that, we need to also prepare our code at $1000 to continue it on the other side. As the cold start vector redirects us here, first we need to go through some initial steps of the official cold reset sequence (so we are now already in C64 mode after the GO64 call):

```
jsr     $fda3
jsr     $fd50
jsr     $fd15
jsr     $ff5b

jsr     $e453
jsr     $e3bf
jsr     $e422

ldx     #$fb
lda     #$30

sei
txs
sta     $01
```

The last instruction also pages out everything here (including the I/O) as we need full access to the RAM again (even up to $feff) to do the relocation.

Now it is the time to do the restoration of those nine bytes and the $ba value, too.

After having that, we relocate the program from $1301 to $0801 (by a simple copying cycle). Also do not forget to set the end of Basic program pointer (at $2d-$2e) to the corresponding ending byte!

Note: to do the relocation well, we firstly need to navigate to any other location (entirely out of the $0801-$feff area!) for the continuation of the remaining code parts (or otherwise the relocating cycle will overwrite itself, and crash!). It is therefore recommended to copy the remaining code parts (see below) into the stack, and jump to there.

Then we page the system back:

```
lda     #$37
sta     $01
cli
```

And we now have yet another job exactly at this point: to also do the rechaining of Basic lines (which should have normally been done by the Basic LOAD command):

```
jsr     $a533
```

We also need to copy the RUN + <CR> characters into the keyboard buffer (at $0277), more exactly these four bytes (it also requires to write $04 to $c6 of course):

```
byte    $52,$55,$4e,$0d
```

Finally we jump onto the warm start by:

```
jmp     ($a002)
```

At this point, the operating system gets the control back from us, and displays the cursor prompt, at where the RUN command gets “magically” typed in… And the program starts (exactly as if it were normally loaded).

Speeding Up the Memory Access

As could be seen, the C64 is actually one of the *slowest* Commodore machines – at least just considering the CPU power. It is getting even worse once it comes to a memory block copy – which must be done by using the CPU power, in a normal cycle, byte by byte. Thus, such simple tasks, like scrolling the screen, may often mean a hard job for this machine.

Normally, a typical inner loop of a copying cycle looks like this:

```
loop    lda     ($xx),y
        sta     ($zz),y
        iny
        bne     loop
```

If no page boundary is crossed, it means at least 5 + 6 + 2 + 3 = 16 machine cycles for each and every byte copied. (If any page boundary is crossed, that is +1 cycle for the LDA, so it may be up to 17 cycles in all.)

A little bit better solution when using this:

```
loop    lda     $xxxx,y
        sta     $zzzz,y
        iny
        bne     loop
```

That is 4 + 5 + 2 + 3 = 14 cycles with no page boundary (or up to 15 with it).

So if 1K of data must be copied, then it needs some 14K-17K (or more) cycles.

On a PAL machine, the time slice during one single frame on screen (both including the visible and non-visible areas) contains only about 1M / 50 = 20K machine cycles within; whereas on an NTSC machine, only about 1M / 60 = 16K cycles. That finally means that one such whole frame time period is only hardly enough to copy 1 kilobyte in memory.

However, if we have (and can detect and use) any of those nice extensions to the stock system, then it can be dramatically sped up. *Let us see a few such cases now:*

The 65816 block copy: if you have a 65816 CPU (for example on the SuperCPU), then you have already got two built-in commands for the automated block copy. They do the same as you would normally do in a cycle, but much faster, since hard-wired to the CPU. That more exactly means a **7 cycles per byte** speed (i.e. more than twice as fast as the above examples). There are two of them: one for upward and the other for the downward direction (in case there would be an overlap between the source and destination blocks).

The upward one is called as MVP (aka “Block Move Positive” on $44 opcode), while the downward one as MVN (aka “Block Move Negative” on $54 opcode), and the syntax of both of them is the same as follows (where $xx is the source and $yy is the destination bank):

MVP (or MVN) $xx, $yy

X register = source address
Y register = destination address
A register = number of bytes to move -1

All registers are the 16-bit ones (so it must be switched to native mode!), and also two things must be kept in mind: first, the Data-Bank-Register will be set to the destination bank, and second, that the above form is only for the Assembler, while it in the memory looks like:

$44, $yy, $xx *(the destination bank comes as first, and the source bank as second!)*

As can be seen, they are capable of copying data anywhere throughout the whole 24-bit address space (up to 16 MB). Even further accelerated if the CPU is on the turbo speed.

The ZP + SP method: this so-called name is the abbreviation of the *Zero Page* (that is sometimes also referred to as *Base Page*) and *Stack Page*. Some platforms have the ability to relocate these two pages into almost anywhere in memory (sometimes even in some other banks). After having done so, you may have a faster access to those spaces through the zero page and stack commands. At least these four platforms are capable of that: C128, C65, DTV and SuperCPU (although the actual settings and behaviours are different on all of them).

This method has some drawbacks, too: on one hand, the zero page addressing still remains limited to 254 bytes instead of the full 256 bytes of the page (since the $00 and $01 locations are part of the CPU itself); and on the other hand, the relocation of the stack makes it a little “dangerous” (thus the interrupts must be disabled for this while, and you should also think of the NMI besides the IRQ).

However, a great advantage is that it can even be used for memory transfer between two banks (which would be especially slow and difficult to do in a normal cycle otherwise).

Even nearly up to **6 cycles per byte** speed can be achieved (or if doing e.g. on the C128 in fast mode, it can be counted as **3 cycles per byte** because of the double speed, and so on). That is undoubtedly rather fast indeed.

On C128, it works only in the native mode (as it is based on the MMU, which is not present in C64 mode). To relocate the zero page, first write the bank number (0-3) into $d508, then write the page number into $d507; to relocate the stack, first write the bank number (0-3) into $d50a, then write the page number into $d509. (These two pages will always use the RAM, and never interfere with ROM and I/O.) In order to perfectly use the ZP + SP method for accessing all memory in all banks, it is also recommended to temporarily disable the RAM sharing (the usage of the common memory portions among all separated banks) by writing 0 into $d506; then after the operation to restore its value, too (the default value is 4 here for the 1K common RAM usage at the bottom, and without that, the operating system sucks); or else only the page numbers will be applied (and bank numbers stay within the BANK 0 instead).

On C65, it is all part of the 4510 CPU. New registers are introduced: one called as Base Page register (or simply B), and the other called as Stack Pointer High Byte. The Base Page register can be accessed via two transfer commands in conjunction with the A register (Accumulator): TBA ($7b opcode) and TAB ($5b opcode). The Stack Pointer High Byte can be similarly accessed via two transfer commands in conjunction with the Y register: TSY ($0b) and TYS ($2b). (The latter ones are also similar to the already well-known TSX and TXS commands, and these two pairs might even handle the low and high bytes of the stack pointer together, once it is switched over to 16-bit mode to use a larger stack; yet it is in 8-bit mode by default, and then behaves more or less like the C128 relocated one.) It seems that both of them are only possible to set within the BANK 0 on this machine.

On DTV, also there are new registers (called Base Page and Stack Base) that can be set through some other special opcodes. (For changing the Base Page: $32 / $AA / LDA #$xx / $32 / $AA; and for changing the Stack Base: $32 / $BB / LDA #$xx / $32 / $BB.)

On 65816, the things are getting more complicated: for the relocation of the stack, it must firstly step into its own native mode again. (Which is not yet needed for the relocation of the ZP, or as called here, the Direct Page.) Thus, we have to temporarily switch to native mode, and then back to emulation mode after the operation (by using the XCE command).

After all, once the ZP + SP pointers are set (according to the actual machine), copying 254 bytes of data looks like this (where the source and destination blocks are page-aligned):

```
        ldx     #$ff
        txs
        dex

loop    lda     $01,x
        dex
        pha
        bne     loop
```

As can be seen, the inner loop is still 4 + 2 + 3 + 3 = 12 cycles per byte here. So it is not that much better… yet. Our final step will be to replace the above loop with speed code:

```
loop    lda     $ff
        pha
        lda     $fe
        pha

        (…)

        lda     $02
        pha
```

Which finally brings that 6 cycles per byte to here. (But do not forget that the $01 and $00 positions are not moved yet, so they still must be “manually” done, apart!)

The REU block copy: the REU (RAM Expansion Unit) has its separated external memory up to 16 MB as well as an integrated DMA controller chip (called REC), which can be used for transferring between the expanded and normal memories, back and forth. One such transfer may embody up to 64K at one go, and during that time the DMA controller replaces the CPU on the bus – which means that it can even be intercepted by a just incoming interrupt at any time (and after the interrupt request having been handled, passed back to the CPU and returning from that to DMA again, the intercepted transfer continues). ***+note:***(5)***

So it must be very friendly – and, even better, incredibly fast: only **1 cycle per byte**!

Nevertheless, if we want to to use it as a block copy method right within the normal memory space, for instance, then we need *two* transfers (once from main memory to REU memory, and once again, from REU memory back to main memory, at the new destination location): so it counts as double then, as **2 cycles per byte** in all (but it is still not so bad).

The REC controller must always be called at 1 MHz (any fast/turbo modes must be turned off before!). Any C64 or C128 machines may have got this expansion by the way.

The DMA block copy: two platforms (the C65 and the DTV) have also got their own DMA controllers onboard, which work similarly to the REU, more or less. (The C65 may have this up to 8 MB, while the DTV up to 2 MB.)

They are even better, since their address spaces embody the main memory, too, so they can do the same job in one single turn. (And they can be called in fast mode, too.)

The VDC block copy: the C128 has its particular (16K or 64K) video memory dedicated for the VDC chip, too. While it is so painfully slow to send any data through its two-byte ports (at $d600-$d601), once the data have already entered the VRAM, then it can be much more quickly moved around within. This is also some kind of DMA transfer after all, but executed by the VDC chip on itself. That also means: it happens independently of the CPU – so you need not to wait for the result, but can do anything else meanwhile.

As a most extreme example, it may even be possible to execute a VDC block copy in the VRAM, while doing an REU copy on the other side – in parallel. (But the VDC chip can only deal with 256 bytes maximum at one go, so it is a little bit complicated then to organize.)

Programming the DMA (the REU and the DTV)

These two ones are pretty well documented, and their programming is easy, so I only give some short summary on them:

REU: it appears at either of the $dexx or the $dfxx I/O areas (depending on actual hardware), and has 11 registers ($dx00-$dx0a). For preparing a simple data transfer, you must first specify the main memory address as a 16-bit value ($dx02-$dx03), the REU memory address as a 24-bit value ($dx04-$dx06) and the transfer length ($dx07-$dx08). After having finished that, also write a command code into the command register ($dx01): generally 0 or 1 (as for transferring *to* or *from* the REU memory; or 2 or 3 for comparing or swapping), but it must also be OR'ed with $80 for the execution (or else nothing will happen), and also with $10 for *instant* execution (or else it will be delayed until writing anything to $ff00).

The actual execution will either be started when writing the command register (if $10 also specified), or when writing to $ff00 (if delayed). The delaying option can be used for changing the memory layout in the meantime (e.g. by paging in and out system ROM and I/O for the “hidden” RAM underneath or other spaces being accessed by the DMA).

DTV: it appears at $d3xx with 32 registers ($d300-$d31f), which must be first made visible by writing $01 into $d03f (and after the operation having been finished, it is also recommended to make them hidden again, by writing $00 into there).

For preparing a data transfer, you must first specify the source and destination addresses, both as 24-bit numbers ($d300-$d302 and $d303-$d305), and the highermost bytes of both of them OR’ed with $40 (as the two highermost bits are indicating 00 or 01 for ROM or RAM); and the transfer length ($d30a-$d30b). You can also specify a source step and a destination step ($d306-$d307 and $d308-$d309), which are normally both $0001; and some other things (like modulo and line length etc.), which are not so interesting now.

The command register is $d31f, and you need to write a $0d command code there for instant execution (but it may also accept several options for delaying and direction changing etc.). It also acts as status register when being read (where bit0 aka $01 signs “DMA busy”).

The DTV has got a separate and dedicated blitter, too, at $d320 with different and additional 32 registers ($d320-$d33f) specialized for graphical data manipulation.

Programming the DMA (on C65)

Well, this third one used by the C65 is still a kind of mystery. There can be only very little description found on the internet, which says unfortunately not too much. (Or at least I have not found the right ones yet…) That ominous writing from ’91 called “Preliminary” has got a short paragraph on this: it mentions the controller is called as **DMAgic** and it has only four registers (at $d7xx address space of I/O): $d700-$d702 write-only to specify an address for some table containing a to-do list (as LB/HB/bank number) and $d703 read-only as status register (where bit7 aka $80 is the “busy” bit). Yet it is not clear exactly what data in that table there must be, and what the controller will do with them at all.

Luckily, we have got some other starting points: there is the DMA command in the C65 Basic v10 dialect, and the following three Kernal calls, all implemented in the system ROM’s somewhere (and eventually the POKE and PEEK functions are also implemented through the DMA) – at least some of the answers must be found there. So I sat before my PC running the MESS emulator, and started to make some disassembly by using the built-in monitor (as if it were done on a real C65 machine, yeah).

The three Kernal calls are actually the same as those already found on the C128 (but for using in context of the DMAgic and with slightly different parametering, of course):

```
jsr $ff74       lda (x),y from bank z
jsr $ff77       sta (x),y to bank z
jsr $ff7a       cmp (x),y to bank z
```

From the virtual jumping table, their absolute addresses can be seen (whence, by withdrawal of the $f26a-$f2bd area, we get their codes):

```
jsr $ff74       jmp $f26a
jsr $ff77       jmp $f28e
jsr $ff7a       jmp $f2b5

$f26a:  php
        tya
        clc
        adc     $00,x
        sta     $03d3
        lda     #$00
        sta     $d702
        adc     $01,x
        sta     $03d4
        stz     $03d5
        lda     #$03
        sta     $d701
        lda     #$d0
        sta     $d700
        plp
        lda     $03cf
        rts

$f28e:  php
        sta     $03cf
        tya
        clc
        adc     $00,x
        sta     $03e1
        lda     #$00
        sta     $d702
        adc     $01,x
        sta     $03e2
        stz     $03e3
        lda     #$03
        sta     $d701
        lda     #$db
        sta     $d700
        plp
        lda     $03cf
        rts

$f2b5:  pha
        jsr     $f26a
        pla
        cmp     $03cf
        rts
```

The very first thing to be noticed is that they are directly relying on those adresses at $d7xx: so it means that we should never forget to apply the two “magic bytes” at $d02f before calling them, or else they might not do anything. (The “danger” about it is the emulator usage for the development process, where the registers are always present at the I/O area.)

The second is that they seem to use some already predefined table(s) “around” $03d0 or so (including that $03cf where the controller must place the one-byte result in some way).

The source of the DMAgic command lists can be found in the Kernal at $f252-f269 (right before the above codes) from where it gets copied to $03ce-$03e5:

```
$03ce: byte    $00
       byte    $00

$03d0: byte    $00
       byte    $01,$00
$03d3: byte    $00,$00,$00
$03d6: byte    $cf,$03,$00
       byte    $00,$00

$03db: byte    $00
       byte    $01,$00
$03de: byte    $cf,$03,$00
$03e1: byte    $00,$00,$00
       byte    $00,$00
```

If we carefully examine and compare the two lists byte by byte, then the structure can easily be recognized: the first byte is the command code (here is set to $00 for copying), followed by the two bytes of the 16-bit transfer length (here is set to $0001), then twice by the three bytes of the 24-bit source and destination addresses (where in both cases the “$03cf in bank 0” appears in context of the other being overwritten by the above subroutines).

This order of the parameters right exactly matches that of the DMA Basic command, the brief description of which also mentions another command code: $03 for filling (where the low byte of the source address holds the value being used to fill the destination area).

The DMAgic chip overview just shortly mentions a list of possible operations in the following order: “Copy (up, down, invert), Fill, Swap, Mix (boolean Minterms)”. (So, if the order matches again, should it mean that we could have three different Copy codes (0-2), then after the Fill (3) would come the Swap and probably some others, too…?)

However, the “Preliminary” section on the DMAgic theme mentions these four provided commands in this following order: Copy (0), Mix (1), Swap (2), Fill (3). Now a little playing around with the Basic command in the emulator suggests this must be the right order.

According to my testing in the emulator, it seems only the Copy (0) and Fill (3) functions are implemented after all; when trying the other two, just nothing happens there.

And yet another final (?) note (or appendix): the last sentence of the previous page would have been meant to be "the end of the story" here… but actually there is one more important thing about it to know. A little bit later (after a little bit more playing around with the disassembly) have I only realized that actually **two** versions of the above codes exist.

And that can be because also there are two versions of the DMAgic chip itself.

The above discussed withdrawal is based upon the earliest and the most common **910111** version of the Kernal ROM (which is the most recommended version to be used, since the MEGA65 also leans upon this one!) that originates from the Rev2B motherboard. So that might be considered as some kind of "standard". This one contains the **F018A** DMA.

However, all of the other Kernal ROM's (so the other five out of the six pieces can be found on the internet) have the other version, which is slightly different, since they originate from some newer revisions of the motherboard (up to Rev5) that contain the **F018B** DMA instead. This newer model uses **one byte longer** tables than the previous (so there will already be now three "meaningless" zeroes at the end of the data instead of two, which seem to be not really used by the software momentarily).

Moreover, they are placed totally elsewhere in memory (at $0120 instead of $03d0, and $015c is used instead of $03cf), and even the sources of the data can be found elsewhere in the ROM. (The latter is even different in most versions…) *Here follow the new tables first:*

```
$0120: byte   $00
       byte   $01,$00
$0123: byte   $00,$00,$00
$0126: byte   $5c,$01,$00
       byte   $00,$00,$00

$012c: byte   $00
       byte   $01,$00
$012f: byte   $5c,$01,$00
$0132: byte   $00,$00,$00
       byte   $00,$00,$00
```

In the latest Kernal known (as version **911001** from the Rev5 motherboard), this sequence can be found at $f29b-$f2b2, followed by the three absolute addresses of the calls:

```
jsr $ff74      jmp $f2b3
jsr $ff77      jmp $f2da
jsr $ff7a      jmp $f304

$f2b3: php
       tya
       clc
       adc   $00,x
       sta   $0123
       lda   #$00
       sta   $d702
```

```
        adc     $01,x
        sta     $0124
        bcc     *+1
        inz
        stz     $0125
        lda     #$01
        sta     $d701
        lda     #$20
        sta     $d700
        plp
        lda     $015c
        rts

$f2da:  php
        sta     $015c
        tya
        clc
        adc     $00,x
        sta     $0132
        lda     #$00
        sta     $d702
        adc     $01,x
        sta     $0133
        bcc     *+1
        inz
        stz     $0134
        lda     #$01
        sta     $d701
        lda     #$2c
        sta     $d700
        plp
        lda     $015c
        rts

$f304:  pha
        jsr     $f2b3
        pla
        cmp     $015c
        rts
```

Thus the final consequence is only that the new version of the table is one byte (actually one zero byte) longer: that is not so dangerous, when programming the DMA chip directly (and building your own tables for this purpose!), you only need to make it one byte longer, too, and therefore stay compatible with both versions at once. (Nevertheless it also means that the Kernal software versions are not interchangeable over the real hardware versions by the way, but it is no problem for the emulator, nor probably for the FPGA clone.)

This newer version of DMAgic similarly only has the Copy (0) and Fill (3) functions implemented; and the other two command codes similarly do nothing.

TDC (and Other Turbo Cards)

Okay, I am writing yet another final chapter here (just to make it be a round 50… er, sorry, 51 pages after all)… may be considered as Appendix (Two) or so.

There exists the "traditional" way of C64 turbo cards, at least since '90, when the first two ones had appeared: the **Turbo Master CPU** by Schnedler Systems in the USA (also called as "Schnedler cart") and the **Turbo Process** in Germany (by Rossmöller) on 4 MHz. Then came the **Flash 8** also in Germany in '92 (as successor of Turbo Process) on 8 MHz. On top of this evolution sits the **SuperCPU** by CMD (Creative Micro Designs) in 1997-2001, both for the C64 and the C128 (as called as SuperCPU64 and SuperCPU128) on 20 MHz.

All of them are based upon the same idea of changing the processor for another one (the first has got a 65C02, whereas the others have a 65816 CPU), together with their own additional RAM and ROM, and everything be placed onto an external cartridge for the expansion port. As a matter of fact, the **Turbo Chameleon 64** is also based upon this idea (but using an FPGA re-implementation of the whole computer and some further extensions).

And there is the **TDC** (or **Turbo Demo Card**), made by Kisiel in Poland 2011, that is an entirely different approach: it does not swap anything for something else, but rather makes an *overclocking* of the *original* motherboard. Or at least of the HMOS chipset of the newer mobos (C64E), since it fits there only, as being an internal daughterboard in the form factor so that can be placed between the sockets and the IC's of the VIC-II and SID (the 85xx ones). These two latter are the bottleneck of Commodore design (also for the C128) that limits the system bus to 1 MHz, while most of the other chips could have been made much faster. And exactly this is what TDC does: it leaves these two at 1 MHz, while overclocks the other ones.

Unfortunately, only a few pieces have been manufactured (although the maker states that he has yet some further future plans); but luckily, I managed to own one of them.

It has got two switchover bits at $d030: bit0 and bit1 (aka $01 and $02). The first bit is for soft-switching between the normal and the turbo operating modes (i.e. in the same manner as the C128, the Turbo Chameleon and the MEGA65 do it, so they can be all programmed in a compatible way in C64 mode). The unit also has a manual switch for this purpose, which overrides the soft-setting when pressed. The machine is in normal 1 MHz mode on power-up, and the 2 MHz mode may be selected by either of these hardware/software ways.

The other bit is for selecting an extra speed (so it only gains an effect once the turbo mode is set in any of the above-mentioned ways) of over 3 MHz, or when the screen is disabled at the same time, too (at $d011), then reaching near some 4 MHz after all. (Care should be taken of this setting, since the C128 has got the VIC-IIe test bit there!)

Once in 2 MHz mode, the good old CPU of our favourite C64 will run a little bit faster than that of the C128 (see below the comparison table for the exact MHz!), but *without* the "VIC-IIe bug" in advance (no garbling artifacts on screen). Yet there are some side effects: everything else is accelerated, too (including the CIA's etc.), thus the IRQ's get more frequent (the cursor starts to blink faster), and disk operations not possible (still the manual switch makes a nice workaround for it somehow, even allowing for fast loaders, if pressed).

The 3 and 4 MHz modes are more quirky, with even more side effects: the $00 and $01 CPU ports are no longer accessible at these speeds, so you cannot change the memory layout until getting back to 2 MHz (which, as considering that the switchover bits are part of the I/O area, too, means that you cannot reach the RAM underneath while being at these higher speeds) and sometimes also artifacts on accessing the colour RAM are reported.

Other than the turbo, the daughterboard has got a socket for a second SID (mapped to $d500 at the I/O) and as well a Covox (the audio outputs of both are redirected for the main).

The $d031 register can be read for a version number (actually $d1), and be written for the Covox output. The $d02f is claimed to contain $f0 (albeit on my unit it is $ff instead). So the common combination of theirs can be used for detection, for example in this way:

```
lda     $d02f
and     $d031
and     #$f0
cmp     #$d0
beq     tdc
```

An important note: there is a serious hardware incompatibility between the fast mode and the presence of the *1541 Ultimate* (I, II, II+ and probably some further cartridges alike, e.g. the Chameleon etc., although they have not yet been tested) that causes an *instant crash* on turning the fast mode on. (If you also detect any of them, please do not use the TDC.)

And finally, as an Appendix (Three) or so, here follows a comparison table on speed among some turbos and the stock systems that I have already measured by now:

*The **maximums** I reached on these systems (with screens and IRQ's turned **off**):*

C64, *PAL / NTSC / NTSC (old) / PAL (Drean):*	0.98 / 1.03 / 1.01 / 1.01 MHz
SuperCPU (VICE), *(everything else as above) (*):*	8.83 / 9.22 / 9.04 / 9.11 MHz
SuperCPU128 (real), *PAL, in C64 / C128 mode:*	8.37 / 8.73 MHz
VIC-20 (VICE), *PAL / NTSC:*	1.11 / 1.02 MHz
TED (YAPE), *PAL / NTSC:*	1.70 / 1.71 MHz
DTV (VICE), *PAL / NTSC (*):*	1.75 / 1.82 MHz
C128, *PAL / NTSC:*	1.88 / 1.97 MHz
TDC, *set to 2 MHz / 4 MHz:*	1.91 / 3.71 MHz
Turbo Chameleon, *set to 6 MHz / maximum:*	5.90 / 14.47 MHz
C65 (MESS), *slow mode / fast mode:*	1.18 / 4.14 MHz
C65 (XEMU), *slow mode / fast mode (**):*	1.20 / 4.19 MHz *(before 2020 fix)*
C65 (XEMU), *slow mode / fast mode (**):*	1.03 / 4.19 MHz *(after 2020 fix)*
C65 (real machine), *in slow mode:*	1.02 MHz ***+note:***(2)***
MEGA65, *in fastest mode (**):*	(between 36 and 56 MHz or so)

(*) Note (1): as can be seen, the SuperCPU is far below its "nominal" 20 MHz in real… as well as the DTV is so below (in this special case at least). (However, I have only checked for the SCPU and the DTV emulations can be found in VICE v2.4 and v3.0, thus on some other real hardware it might be slightly different…) ***+note:***(3)***

(**) Note (2): Measuring for MEGA65 in 40 MHz (or before called as 48 MHz) fast mode varies between 36 and 56 MHz, depending on actual revision or version (as well as the different versions of the XEMU emulator included here). That is a matter of development still, so it might be changing on in future. ***+note:***(3)***

The above list is far from being final or complete at the moment, but I will not be refreshing it here any more, as it is meant indicative only. (A more recent and updated list should be found in the Rosetta user manual later and be maintained there.)

About the Author

This writing can be found on my website on the internet:

http://istennyila.hu/dox/cbmcode.pdf

This entire programming guide or handbook, in conjunction with my *MemTest64* and *SDOS* projects, initially started as an independent part of my *Rosetta Interactive Fiction* project. (As I needed some kind of utilities, and since I had not found any, I have had to make it by myself.) However, you can also freely use or apply it, of course.

The MemTest64 and SDOS codes are Public Domain: open-source and freeware.

Here are some direct downloading links to them:

http://istennyila.hu/stuff/memtest.zip

http://istennyila.hu/stuff/sdos.zip

MemTest64 *project homepage:*

http://istennyila.hu/memtest64

SDOS *project homepage:*

http://istennyila.hu/sdos

Rosetta Interactive Fiction *project homepage:*

http://istennyila.hu/rosetta

(On opening them please click onto the greeting images for entering the main page!)

Robert Olessak (2012-2017)

As Epilogue: *There have been some fixes since 2017 (being made in 2020 when I slightly revised my article) and signed as* ***+note:***(…)*** *inside the text:*

+note:(1):*** The "Ultimate family" was completed with the *Ultimate-64* (abbreviated as *U64* here) in 2018 (also there is a new *Turbo Chameleon V2*).

+note:(2):*** It is already documented in the *"C65 Preliminary"* that the 4510 CPU in slow mode applies a kind of "dummy cycles" as a compensation for a better 65xx compatibility, so the actually measured value must be about 1.02 MHz in real. Seemingly this feature is not emulated by the MESS emulator (and hence the 1.18 MHz), but it has been fixed in MEGA65 and the XEMU emulator since then.

+note:(3):*** There have been some speed measurements being done on some of the above-mentioned hardware since then (like SuperCPU128, MEGA65 and even a real C65 prototype machine!), thus the table of results has been slightly changed.

+note:(4):*** SDOS 2017 has not been made (as under this name never will). Instead, what formerly was SDOS 2016 is renamed now as SDOS v1.0, and newer versions have been made (up to v1.3 momentarily). V2 is under development (maybe coming soon this year).

+note:(5):*** Unfortunately, this seems containing some false info here, as IRQ is not able to intercept an actual DMA operation in transfer (but rather is delayed until the operation ends); I still don't know what the case on an NMI is (probably related to IRQ).

There has been an upload made at CSDb.dk out of this text (still before these annotations being made) that can be found here:

http://csdb.dk/release/?id=155862

Have a good fun!

Robert Olessak (2012-2020)