commodore64.se – Datorernas folkvagn
Den största skandinaviska wikin om VIC-20, C64, C128!

commodore64.se: Forum | Wiki

RSS...

Från commodore64.se - Datorernas folkvagn

Commodore 128 assembly programming

Innehåll

[göm]

Introduction

Have you programmed in assembly on the Commodore 64? Are you interested in learning how to program in assembly on the Commodore 128? If the answer to these questions is yes, this is the page for you. If you haven't programmed in assembly on the Commodore 64, I can recommend the book Machine Language for the Commodore 64 and Other Commodore Computers by Jim Butterfield, which contains prerequisites for understanding this page.

This page gives an overview of the differences between C64 and C128 when programming in assembly. I have focused particularly on issues that are poorly described or not described at all in books about the Commodore 128 that I have read. Since this page is only intended as an overview, it does not contain enough information for being able to program the Commodore 128 in assembly. I recommend that you read this page together with the book "Commodore 128 Programmer's Reference Guide" to get a complete description of how to program the Commodore 128 in assembly. I have read the German translation of the book (called "Das C128 Buch"). The book "Das C128 Buch" contains a lot of obvious errors (many of them may be problems in the translation from English to German) but it is still very good and comprehensive (869 pages). Since most things are the same on the Commodore 64 and the Commodore 128, the book is valuable even if you just want to program on the Commodore 64. For example, there is an interesting example on how to program a split-screen on the Commodore 64 using raster interrupts. That example also contains a scrolling text.

Another book that I recommend is "Mapping the Commodore 128". It is a book that describes what every routine in the ROM does and what every memory location in the RAM is used for, which makes it extremely good as a reference book (but it is not good for reading from beginning to end). The book consists of 689 pages.

As you read this page, you will probably notice that one major difference between the Commodore 64 and the Commodore 128 is that on the Commodore 128, the BASIC interpreter and the Kernal write to various registers when interrupts occur so if you write directly to these registers, the values will be overwritten the next time an interrupt occurs. Because of this, you either have to write to shadow registers instead of to real registers or you have to disable the BASIC interpreter and the Kernal from writing to these registers (this disabling will cause limitations to the BASIC functionality).

Unless stated otherwise, all information on this page applies only to the Commodore 128 in Commodore 128 mode (the default mode of the C128).

If you want system information about your Commodore 128, you can use the program "SYSTEM INFO R5A" that I have implemented. It can be found in this archive file. It gives you information about for example ROM versions, VDC chip version, the amount of VDC RAM and the version of the SID chip (6581 or 8580).

If you have any comments on this page, you are welcome to write to me. Register at the forum and send me (Commodorianen) a private message.

Documentation

- "Commodore 128 Programmer's Reference Guide"This is an invaluable book if you want to learn how to program in assembly on the C128.
- "Mapping the Commodore 128"This is an invaluable reference book if you want to program in assembly on the C128. It describes what every routine in the ROM does and what every memory location in the RAM used by the operating system is used for.
- Memory maps for Commodore 128(and some other Commodore related documentation)(not as detailed as in "Mapping the Commodore 128")
- Here you can find a commented ROM disassembly.]
- "The Transactor magazine"This publication contains several interesting C128 articles by the Commodore guru Jim Butterfield and other people.
- The C128 Basic Interpreter
- "Commodore 128 Bedienungshandbuch"This book was included with Commodore 128 computers sold in Germany. It is very different and much more detailed than "Commodore 128 System Guide" so read this book instead if you understand German. The book is pretty basic and does not give any insight in assembly programming on the Commodore 128 but it is good to read to learn the fundamentals of the Commodore 128.
- "Commodore 128 System Guide"This book was included when you bought a new Commodore 128 computer. It is pretty basic and does not give any insight in assembly programming on the Commodore 128 but it is good to read to learn the fundamentals of the Commodore 128.
- Commodore 128 Alive!contains information about the C128, C128 programs and a C128 discussion forum

Assemblers

A really good assembler for the Commodore 128 is Power Assembler (Buddy).

- Power Assembler (Buddy) version 8.2 for the C128
- Power Assembler (Buddy) version 8.2 for the C64
- Documentation for Power Assembler (Buddy) version 8.2 for the C128 and the C64(includes a brief tutorial to be run on the C64/C128)
- Power Assembler (Buddy) version 10 for the C128 and the C64The main differences compared to version 8.2 are that version 10 supports macros and having two 40-column windows in EBUD when the 80-column screen is used.
- Documentation for Power Assembler (Buddy) version 10 for the C128 and the C64
- Programs I have implemented for the Commodore 128Power Assembler version 8.2 source code is included for assembly language programs.
Note that you must load with DLOAD"BUD". Using LOAD"BUD",8 does not work. If you know why, please tell me!

Bug in Power Assembler version 8.2 for the C128 (I have not tested whether it exists in version 10): When I assembled using the .BAS pseudo-op, I wanted some data (defined using .BYT) after the end of the assembly source code to be put at a specific address ($2000) that was after the assembly source code. It should be possible to do this using .ORG or * =... before the .BYT statements. However, when I did this, the data was put at address $1FFE instead of $2000 in memory (an offset of 2 bytes). I managed to solve this by defining that the data should be put at $2002 instead. When looking at the symbol table generated at the assembly, it looked like the data had indeed been put at $2002 as defined but when looking at the generated data in the C128 machine code monitor it could be seen that the data had in reality been put at $2000 as intended.

- Double-Ass(another assembler for the C128 including brief documentation, this one I have not tried myself but it seems less powerful than Power Assembler)

Memory management

Both the Commodore 64 and the Commodore 128 have a 16-bit address bus and can therefore address 64 kB of memory at the same time. For both of the computers, ROM memory can be placed at the same memory locations as RAM memory. When this is done, a write operation to a memory location writes to the RAM while a read memory reads from the ROM. It is then said that the RAM lies under the ROM and this is sometimes also referred to as "hidden" RAM. If a ROM is switched out, both read and write operations operate on the underlying RAM.

For the Commodore 64, the register at address $0001 is used to decide which ROMs that should be switched in and which ROMs that should be switched out. In total, the Commodore 64 has 64 kB of RAM and 20 kB of ROM memory.

The Commodore 128 has more sophisticated memory management than the Commodore 64. The C128 has a separate chip called the MMU (Memory Management Unit) for memory management, which the Commodore 64 does not have. Using the MMU, it is possible to set four different pre-set memory configurations. The pre-set memory configurations can only be defined when the I/O ROM is switched in but once the pre-set memory configurations have been defined it is possible to switch between them regardless of how the memory is configured. The first three of the four pre-defined memory configurations are used by the operating system (the two first ones are used by the BASIC CHRGET routine and possibly at more places as well). If you try changing one of these three pre-set memory configurations, BASIC may no longer work correctly (you may end up in the machine code monitor). You can however always set the memory configuration directly by writing to memory location $FF00 without using pre-set memory configurations.

A memory configuration defines which ROMs that should be switched in and which ROMs that should be switched out. The Commodore 128 has two RAM banks (bank 0 and bank 1), each consisting of 64 kB RAM, and the memory configuration also defines which of the two RAM banks that should be switched in (the other one is switched out). When using BASIC, RAM bank 0 is used for program code and RAM bank 1 for variables. That variables are put in another RAM bank than BASIC program code makes it easier to mix BASIC and assembly code than on the Commodore 64 where it is easy to destroy program code unless it is carefully thought about which address the Start-Of-Variables (SOV) pointer points at after a LOAD.

The MMU also makes it possible to define if memory should be shared between the two RAM banks. If memory is shared, it can be defined how much memory that is shared and if memory should be shared at the bottom or at the top of the banks or both. Sharing memory means that if bank 1 is switched in and a memory location in a shared area is accessed, it is the memory location in bank 0 that is accessed instead of in bank 1.

The VIC chip can access 16 kB of memory at once (both in Commodore 64 and in Commodore 128) meaning that it can point at four different places in a 64 kB RAM bank. The MMU in the Commodore 128 is used to define which of the two RAM banks that the VIC chip should use. This can be defined independently from the setting made in the MMU regarding which RAM bank that is switched in for the CPU to see.

The MMU also makes it possible to redirect page 0 (address $0000-$00FF) and page 1 (address $0100-$01FF) in memory. (One page contains $0100 addresses.) This means that if the software tries to access a memory location in these two pages, the access is redirected to a memory location where the lower 8 address bits are the same but the higher 8 address bits are set to another value that has been defined by the user in a register in the MMU. Page 0 contains a lot of variables used by the operating system and page 1 contains the processor stack so the possibility to redirect these pages makes it possible to quickly switch between different sets of variable values and processor stacks.

Of course, the memory map for the Commodore 128 is different from that for the Commodore 64. For example, the BASIC text normally begins at $1C01 instead of at $0801. However, if the BASIC command GRAPHIC is used to define one of the graphics modes, the start of BASIC text is moved to $4001 (under BASIC ROM). Furthermore, the BASIC and machine code monitor ROM ($4000-$BFFF) take up 4 times as much ROM memory as the BASIC ROM on the Commodore 64 ($A000-$BFFF). The memory area $C000-$CFFF that is often used for machine code programs on the C64 is on the C128 used by the screen editor ROM that is a part of the Kernal ROM. On the C128, there are some areas that are particularly suitable for machine code programs. For example, $1300-$1BFF is free. If the tape buffer, RS-232 buffers and BASIC sprite definition area are not needed, $0B00-$0FFF can also be used. In zero page, $FA-$FE are free but $FA is unsafe to use due to a bug in the Kernal. These are some of the major differences in the memory map.

VIC features and bugs

The 8564-VIC graphics chip in the Commodore 128 has two extra registers that the 6567-VIC graphics chip in the Commodore 64 does not have. These two extra registers are used for new keys on the keyboard (more columns in the keyboard matrix) and 2 MHz mode. These features are described at other places on this page. It might seem a bit strange that the 2 MHz register is located in the VIC chip considering that the VIC chip does not support 2 MHz mode (it results in garbage on the screen) but this is the way it is.

The two versions of the VIC chip are not interchangeable. You cannot put a 8564-VIC chip in a C64 or a 6567-VIC chip in a C128. Both the 8564-VIC and the 6567-VIC are often referred to as VIC-II (8564-VIC is sometimes called VIC-IIe where "e" means enhanced). This is because they are successors of the VIC (VIC-I) chip that existed in the Commodore VIC 20 computer and predecessors of the VIC-III chip in the Commodore 65. On this page the 8564-VIC chip is often simply referred to as VIC.

With the 8564-VIC chip, you can see vertical lines going through the middle of each of the columns of the screen. When I first used a C128, I thought that there was something wrong with it but I then found out that this bug exists on all C128s. However, it is more visible on some C128s than on others. This depends on the revision of the 8564-VIC chip. The bug also to a certain degree exists with the 6567-VIC chip in the C64 but there it is much less visible. Another bug in the 8564-VIC chip is that when using raster interrupts, there can sometimes be lots of white dots flashing around. This is visible in some games. The just mentioned bugs in the 8564-VIC chip can be pretty annoying and therefore if you want to play C64 games a lot, I would recommend using a real Commodore 64 rather than a Commodore 128 in Commodore 64 mode.

So, what about graphics features? Are there any differences in that area between the 8564-VIC and the 6567-VIC? The answer to this question was long believed to be no. However, just around the turn of the century (from 20th to 21st) it was found that the test bit in register $D030 (the register used for 2 MHz mode) does indeed hide unknown graphics modes (something that was suspected several years earlier). It was found that as long as this bit is set, one raster line is skipped per clock cycle. This can be used for increasing the screen refresh frequency, which causes a more stable picture, especially for PAL C128s since PAL uses 50 Hz (NTSC uses 60 Hz). The test bit can also be used for more advanced techniques as described in the following sections.

On PAL systems, skipping an odd number of raster lines not only increases the screen refresh frequency but also leads to new colours in addition to the 16 usual ones. This is used in the C128 demo "Risen from Oblivion". It is impressive to see the frogs in lots of different green colours in that demo. I really recommend that demo, which has one VIC part and one VDC part. It is the best C128 demo there is (and one of the very few there is as well).

In 2007, it was found that skipping raster lines makes it possible to accomplish interlace mode, which doubles the vertical resolution, although it causes some flickering. The experiment was made using an NTSC C128 but it is probably possible to do this on a PAL C128 too.

The test bit tricks all have in common that they only work with real C128s, i.e. not with emulators, and that they only work together with certain monitors and TV sets. The VIC part of "Risen from Oblivion" for example works with most Commodore 1084 monitors (I have however seen one person reporting that it did not work with his 1084 monitor).

VIC graphics and character programming

One difference between the Commodore 64 and the Commodore 128 when it comes to graphics is that the C128 has built-in support for split-screen mode. When split-screen mode is enabled, it means that there is a vertical division of the screen into two different areas. One of the areas uses bit-map mode or multi-colour bit-map mode while the other area uses standard character mode. This is possible to do also on the C64 but there is no built-in support for it so you need to write your own raster-interrupt code to do it there.

The screen editor (which is a part of the Kernal) in the Commodore 128 is raster interrupt-driven in order to make split-screen mode possible. This is a difference compared to the Commodore 64 where the Kernal uses timer interrupts instead of raster interrupts. The C128 screen editor reads the value of so called shadow registers to update actual registers when raster interrupts occur. This means that when the screen editor is enabled, registers that are used to enable/disable graphics modes or set pointers to screen memory, beginning of character data or bit map data (this concerns registers $D011, $D016 and $D018) should not be written to directly. If they are written to directly the written value will just be overwritten by the screen editor when the next raster interrupt occurs. Instead, shadow registers should be written to.

Let us for example assume that we have a split-screen with standard or multi-colour bit-map mode at the top of the screen and standard character mode at the bottom of the screen. Then, we will get two raster interrupts per screen update. We will get the "normal" interrupt at the top of the screen that we always get but since we are in split-screen mode, the raster compare register is reprogrammed when the interrupt occurs so that we will also get an interrupt further down on the screen where we want the split to occur. If we look at the pointer to the screen memory only in order not to make this example too complex (we could also look at other pointers and enabling/disabling of different modes), what happens is that at the interrupt occuring at the top of the screen, bits 7-4 in register $D018 are set equal to the value of bits 7-4 of shadow register $0A2D. At the interrupt occuring further down on the screen, bits 7-4 in register $D018 are set equal to the value of bits 7-4 of shadow register $0A2C. This example shows that different shadow registers are used for pointers used in (multi-colour) bit map mode and in multi-colour/standard character mode even though they are mapped to the same actual registers at different points in time when interrupts occur.

As mentioned in the previous paragraph, there is one "normal" interrupt occuring once per screen update and if split-screen mode is enabled, a second interrupt occurs as well further down on the screen during each screen update. However, it is only during the "normal" interrupt that the Kernal operations occuring at an IRQ are done (e.g. checking keyboard, blinking cursor, updating clock etc.). Whether it is a "normal" interrupt or not is indicated by the screen editor in bit 7 of register $D011. If that bit is equal to 1, it is a "normal" interrupt, otherwise not.

If you do not want to use shadow registers, you can turn off the screen editor by writing $FF to memory location $00D8. Then, you can program graphics in exactly the same way as on the Commodore 64, i.e. by writing to the actual registers directly. The registers have the same addresses as on the Commodore 64.

For both the Commodore 64 and the Commodore 128, the colour RAM occupies the area from address $D800 to $DBE7. However, the Commodore 128 has as mentioned earlier two 64 kB RAM banks and it is possible to select which of these two banks that contains the colour RAM seen by the CPU or by the VIC chip. Bit 0 in the register at address $0001 defines which colour RAM bank that is seen by the CPU while bit 1 in the register at address $0001 defines which colour RAM bank that is seen by the VIC chip. This makes it possible to quickly change the colours of a whole screen. The CPU can define the colours in a colour RAM bank that is not seen by the VIC chip and when it wants the colours to appear on the screen, it just switches in the colour RAM bank it has written to so that the VIC chip can see it. (Bits 0 and 1 in the register at address $0001 are on the Commodore 64 used for switching in or out BASIC or Kernal ROM, which on the Commodore 128 is taken care of by the MMU.) The switching of colour RAM bank seen by the VIC chip can be used to slightly improve some of the advanced VIC graphics modes. For example, in FLI mode, the colour RAM bank can be switched every fourth raster line and in IFLI mode it can be switched every frame. I have come up with this idea for improvement myself. I don't know if anyone else has had the same idea before and if it in that case has been tested in practise. Note that this feature is only possible in C128 mode.

For the Commodore 64, character ROM is only accessible by the VIC chip in video banks 0 and 2 (the VIC chip can only see 16 kB of RAM while there are 64 kB of RAM in total leading to that there are four possible video banks). The Commodore 128 does not have this limitation. Character ROM is accessible in all video banks unless character ROM has been switched out explicitly using bit 2 of the register at address $0001 (if the screen editor has not been disabled by writing $FF to memory location $00D8, bit 2 of the shadow register at $00D9 has to be used instead).

VIC sprite programming

When an IRQ occurs, one of the things that the BASIC part of the IRQ routine does is to read the values of the shadow registers $11D6-$11E6 and write them into the actual registers $D000-$D010 that control the positions of sprites on the screen. This means that it is no use to write to the registers $D000-$D010 to set sprite positions since the written values will be overwritten at the next IRQ. Instead, the shadow registers at $11D6-$11E6 should be written to in order to set the positions of sprites. If you want to write to the actual registers directly, this can be done by setting bit 0 of the system variable at address $0A04 equal to 0. This tells the Kernal that BASIC has not been initialised and therefore the BASIC part of the IRQ routine will not be executed. Because of this, some BASIC commands, e.g. SOUND, PLAY and SPRITE, will not work.

The sprite pointers are on the C128 just as on the C64 normally placed at $07F8-$07FF. However, on the C128, these pointers have pre-defined values pointing to a block of memory at $0E00-$0FFF that is reserved for sprite data. Since Power Assembler uses a part of this memory ($0F00-$1000), you will probably want to change the values of the sprite pointers. The C64 does not have pre-defined values for the sprite pointers.

The sprite pointers are in the C64 and the C128 always located as the last 8 bytes of the 1 kB chunk of screen memory. If you on the C128 use the BASIC command GRAPHIC to go to one of the graphics modes, the screen memory uses addresses $1C00-$1FFF instead of $0400-$07FF. This means that addresses $1FF8-$1FFF are used for sprite pointers instead of addresses $07F8-$07FF.

Sound and SID chip version

Sound is programmed in exactly the same way on the Commodore 128 as on the Commodore 64.

In contrast to what many people believe, most Commodore 128s contain the older version of the SID chip called 6581. My two C128s both contain that version and also other people I have been in contact with have checked the SID chip version of their C128s have that version. The only C128 model I know of that contains the newer version called 8580 is the Commodore 128 DCR (the metal case model that was mainly sold in North America). However, it is possible that late manufactured versions of other C128 models might also contain the 8580 version although I have not encountered any so far. If you want to check the version of the SID chip in your Commodore 128, you can use my program "SYSTEM INFO" that you can find under Introduction. The two versions of the SID chip are not interchangeable without adding or removing other electrical components due to different voltages.

Kernal

The same 39 Kernal calls as on the Commodore 64 are also available on the Commodore 128. A few of these calls differ from the calls made on the Commodore 64. According to "Das C128 Buch", it was necessary to make these changes considering some special features of the C128, e.g. 40-column mode and 80-column mode.

In addition to the 39 Kernal calls that also exist on the Commodore 64, there are 19 new Kernal calls that are Commodore 128 specific. There are for example calls for accessing or jumping to memory locations in a selectable RAM bank (the C128 has two RAM banks each consisting of 64 kB as mentioned before), for going to C64 mode, for booting from an autostart floppy, for switching between 40-column and 80-column modes, for programming function keys and for outputting a string of data.

Except for the official Kernal calls, there are numerous other useful routines in the Kernal ROM, the screen editor ROM and the BASIC interpreter ROM. In particular, it can be mentioned that there are a number of useful routines in the screen editor ROM that is located at $C000-$CFFF. Some of the screen editor routines are called

when the Esc key followed by another key is pressed but the routines can also be called directly from an assembly program. The routines are not described in "Das C128 Buch" but there are various other books that describe at least some of the routines. For example, the "Commodore 128 Tips & Tricks" book (in German) describes some useful routines.

RS-232

The main difference between programming the User Port RS-232 interface (device number 2) on the C128 compared to the C64 is that on the C128 a fixed range of memory addresses is always reserved for the 256-byte output and input buffers. On the C64, on the other hand, the buffers are allocated in the end of BASIC text memory when an RS-232 channel is opened and de-allocated when it is closed. Therefore, on the C64, when an RS-232 channel is closed, the buffer contents are lost and when opening or closing an RS-232 channel, all BASIC variables are cleared. This is not the case on the C128.

The memory addresses used for RS-232 system variables in zero-page are the same on the C128 as on the C64. However, the RS-232 system variables that are not placed in zero-page (e.g. the control register and the command register) are placed at different memory addresses on the C64 and the C128.

Return from interrupt

Both on the Commodore 64 and on the Commodore 128, if you want to execute your own code when an IRQ occurs, you do this by changing the IRQ vector at $0314/$0315 to point to your own interrupt routine. When your own interrupt routine has finished executing, it should then jump (JMP) to the address $FA65 (that was originally stored in the IRQ vector at $0314/$0315) so that the things the Kernal usually does when an IRQ occurs are still done. However, when you use more than one raster interrupt per screen update you probably only want to jump to the Kernal interrupt routine once per screen update. Otherwise, the parts of the Kernal that are handled at interrupt level will be handled too often resulting in strange effects such as the cursor blinking faster than usual. Therefore, there is sometimes a need to be able to return from an interrupt without jumping to the Kernal interrupt routine. The following code should be used for this:

```
PLA STA $FF00 PLA TAY PLA TAX PLA RTI
```

The red lines above should be added for the Commodore 128. For the Commodore 64, these lines should not be present. What the two red lines do is to load a byte from the stack and then store it in the MMU configuration register. This byte was read from the MMU configuration register and stored on the stack by the Kernal when the interrupt occurred.

I have found that returning from an interrupt like explained above does not work together with BASIC (you end up in the machine code monitor or the computer freezes) so do this only when you are programming in assembly! The only way to return from an interrupt that I have found working together with BASIC is to jump to the original Kernal interrupt routine (jumping to address $FA65). However, as written above, this leads to strange effects if you have more than one interrupt per screen update (sprites will move too quickly, music will play too quickly, the cursor will blink too quickly etc.). If you know how to solve this, please tell me!

Keyboard

On the Commodore 64, address $00CB is used to read which key that has been pressed for most keys on the keyboard. On the Commodore 128, it is instead address $00D4 that is used for this purpose. The coding of which key that has been pressed is not the same for the Commodore 128 as for the Commodore 64. The value for "no key pressed" also differs. Furthermore, the Commodore 128 has more keys than the Commodore 64. The easiest way to see which value that corresponds to a certain key is to keep a key pressed and read the value of $00D4.

On the Commodore 64, address $028D is used to read whether SHIFT (SHIFT-LOCK), C= or CTRL has been pressed. The corresponding address on the Commodore 128 is $00D3. On the Commodore 128, there are two more bits that are used for the ALT and CAPS LOCK keys. CAPS LOCK is replaced by ASCII/CC on Swedish keyboards and ASCII/DIN on German keyboards. (CAPS LOCK/ASCII CC/ASCII DIN can also be read through bit 6 at address $01).

The 40/80 DISPLAY key can be read from bit 7 in register $D505 (an MMU register). On both the Commodore 128 and the Commodore 64, NMI (Non Maskable Interrupt) is used to detect that the RESTORE key has been pressed.

The extra keys on the Commodore 128 keyboard can be used also in Commodore 64 mode as is shown in this assembly code example.

On the Commodore 128, it is possible to change the key definitions, i.e. you can define which Commodore ASCII character (0-255) that a press of a certain key on the keyboard should result in. For international keyboards, there are four key definition tables in the Kernal; one normal (also used when ALT is pressed), one when SHIFT is pressed, one when C= is pressed and one when CTRL is pressed. If you have a national keyboard (with an ASCII/CC or ASCII/DIN key instead of a CAPS LOCK key), there are three additional tables that are used when the ASCII/CC or ASCII/DIN key is pressed; one normal, one when SHIFT is pressed and one when C= is pressed. When CTRL or ALT is pressed, the same tables are used no matter if the ASCII/CC or ASCII/DIN key is pressed or not so there are no additional CTRL or ALT tables in the Kernal for national keyboards. The tables are located between $FA80 and $FBE3.

For international keyboards, there are five vectors that point to the start of the corresponding key definition tables; one normal, one when SHIFT is pressed, one when C= is pressed, one when CTRL is pressed and one when ALT is pressed (the one when ALT is pressed points to the same memory location as the normal one). These vectors are located at $033E-$0347. If you have a national keyboard, there is one additional vector at $0348-$0349 that points to the normal key definition table when the ASCII/CC or ASCII/DIN key is pressed. For national keyboards, the Kernal at interrupt level checks whether the ASCII/CC or ASCII/DIN key is pressed or not. When it detects that the key has been pressed, it uses the vector at $0348-$0349 instead of the vector at $033E-$033F for the normal keyboard definition table. Furthermore, it uses the same vectors as before for SHIFT and C= but it changes the values of the vectors so that they point to the national keyboard definition tables for SHIFT and C=. The values of the CTRL and ALT vectors are kept the same as before.

If you want to change key definitions, then copy one or more tables to RAM memory without any overlying ROM memory (by default, this means a memory location with an address lower than $4000). Then, modify entries in the table(s) and change the vector(s) to point to the table(s) in RAM. If you have a national keyboard and the ASCII/CC or ASCII/DIN key is not pressed, you have to disable the Kernal check of whether the ASCII/CC or ASCII/DIN key has been pressed or not. Otherwise, the Kernal will just overwrite the values you have written to the vectors at the next interrupt. You do the disabling by setting bit 7 in the memory location at address $0AC5 equal to 1. One side effect of doing this is that if you do it when the ASCII/CC or ASCII/DIN key is not pressed and afterwards press the key, you will get the same effect as when pressing the CAPS LOCK key on an international keyboard, i.e. letter keys will be SHIFTed while numbers will not be SHIFTed.

I have got information about changing key definitions from appendix J of Commodore 128 Bedienungshandbuch (there you can find the addresses of the key definition tables but one vector is missing and disabling of Kernal check for ASCII/DIN key is also missing) and from chapter 7 "Rund um die Tastatur" in Commodore_128 Tips & Tricks (the addresses of the key definition tables are wrong in this book). Since both of these books are a bit erroneous and contradictory to each other regarding this functionality, I have also had to experiment myself to see how it really works. Please note that this information does not exist in Commodore 128 System Guide (which is the English equivalent of Commodore 128 Bedienungshandbuch) nor does it exist in Commodore 128 Programmer's Reference Guide/Das C128 Buch. Commodore 128 Bedienungshandbuch is not a direct translation of Commodore 128 System Guide from English to German. There is a big difference in the contents. Personally, I think Commodore 128 Bedienungshandbuch contains much more detailed information than Commodore 128 System Guide. For example, in Commodore 128 Bedienungshandbuch, there is detailed information about memory management and CP/M is described in much detail along with all CP/M commands in a separate book.

2 MHz mode

The system clock frequency for the Commodore 64 and the Commodore 128 normally is 1 MHz (slightly less or more depending on if it is a PAL or an NTSC machine). On the Commodore 128, there is a possibility to double the clock frequency to 2 MHz.

Unfortunately, the 8564-VIC chip cannot handle a clock frequency of 2 MHz, which results in garbage on the screen. It may therefore be a good idea to blank the screen by setting bits 4 and 7 of register $D011 to 0 before changing to 2 MHz in order not to see the garbage on the screen. Another alternative is to use raster interrupts so that 2 MHz is only used while the electron beam is outside of the visible screen, i.e. when it is in the lower or upper border. This method is for example used in the game Test Drive II when run in C64 mode on a C128. The advantage of this method is that characters or graphics will still be visible while the disadvantage is that the clock frequency will not be 2 MHz all the time.

The 80-column 8563 chip (which requires an RGBI monitor) can handle a clock frequency of 2 MHz.

The 2 MHz mode can also be used in Commodore 64 mode on the Commodore 128. The same limitation applies as in Commodore 128 mode, i.e. the VIC chip cannot handle such a high frequency, which results in garbage on the screen.

Z80 CPU

In addition to the 8502 CPU, the C128 also contains a Z80 CPU. The Z80 is used in CP/M mode except at Kernal routine calls where the 8502 is temporarily switched in in order to avoid having duplicate Kernal routines in the ROM. The C64/C128 BASIC and Kernal only use the 8502 CPU but although it is very rarely done it is possible to write a program that uses the Z80 from C128 mode. The following paragraphs give a brief description about how to do this. How to program in CP/M mode is not covered here.

Because of the pipelined architecture of the 8502, a 1 MHz 8502 is approximately comparable in speed with a 4 MHz Z80. However, on the C128, the Z80 only runs at 4 MHz half the time giving an effective clock speed of 2 MHz. This means that the Z80 is only about half as fast as the 8502 (only about a fourth as fast when the 8502 is in 2 MHz mode). Despite of this, there might be situations where you would want to use the Z80 from C128 mode. One reason would be if you are short of memory space and speed is not that important. This is because the Z80 has a bigger and more powerful instruction set than the 8502 and it is therefore probably sometimes possible to do things using less memory on the Z80 (I haven't made any comparisons so I can't say this with certainty). There are for example instructions for 16-bit arithmetics and for copying blocks of data. However, a drawback is when you want to do accesses to I/O registers because that requires more instructions on the Z80 than on the 8502 since you have to use the IN and OUT instructions. See the following example:

```
LD A,0 ;PUT 0 (BLACK) IN ACCUMULATOR LD BC,$D020 ;PUT $D020 IN REGISTERS B AND C, NOTE HOW THE Z80 CAN USE 2 8-BIT REGS AS 1 16-BIT REG OUT
(C),A ;SET BORDER COLOR, OUT MUST BE USED RATHER THAN LD FOR I/O REGS TO AVOID BLEED-THROUGH TO UNDERLYING RAM
```

Another reason would be if you are more familiar with the Z80 than with the 8502, for example if you have programmed on the Sinclair ZX Spectrum before. A third reason would be if you simply think it would be cool to do something that not so many people have done before. The last reason was the one that appealed most to me ;) .

How do you switch in the Z80? In RAM bank 0, there is an 8502 routine at $FFD0 to switch to the Z80 and at $FFE0 there is a Z80 routine to switch in the oposite direction. The routines have been copied there by the Z80 boot ROM at start-up. When the routine at $FFD0 is called, the Z80 wakes up where it was switched out in the routine to switch in the opposite direction (at $FFEE to be precise). Where the Z80 wakes up, there is a RST 8 instruction, which causes CP/M to boot. If you want to run your own Z80 program instead, it is a good idea to change the instruction to a JP instruction to jump to your own code.

The 8502 routine at $FFD0 sets the RAM bank to RAM bank 0. In RAM bank 0 with the Z80 enabled, the Z80 ROM (containing reset and CP/M boot code) is seen at address $0000-$0FFF. There is a trick to avoid this. According to most literature, RAM bank 2 is identical to bank 0. However, this is not true when the Z80 is active. If you select RAM bank 2, you will instead get RAM at $0000-$FFFF so when you write your own Z80 programs it might be a good idea to change the routine at $FFD0 to set RAM bank 2 instead of RAM bank 0. You can of course also select RAM bank 1 but since Power Assembler uses that a lot, it might be difficult.

When the Z80 processor is active with the MMU set so that there is I/O at $D000-$DFFF and RAM bank 0 or 2 in the rest of the 64 kB of address space, color RAM is at $1000-$13FF and NOT at $D800-$DBFF where it is on the C64 and on the C128 with the 8502 active. $1000-$13FF is a part of the I/O space so just as for $D000-$DFFF, "OUT (C),A" should be used to write a byte and "IN A,(C)" to read a byte.

Of the three interrupt modes that the Z80 supports, the C128 supports the two modes IM 1 and IM 2 (not IM 0). Personally, I think IM 1 is easiest to set up. With IM 2 you have to fill 257 bytes with the same 16-bit address to the interrupt routine. You also have to set the I register to the base of the 257-bytes block. With IM 1, on the other hand, you know that the Z80 will start executing at $0038 when the interrupt occurs. Don't forget that with the Z80 you have to enable interrupts again in the interrupt routine (using EI). Otherwise, you won't get any more interrupts.

There are some things you can only do with the 8502 but not with the Z80. There is an NMI input on the Z80 but it is connected to ground so this means that you can't get interrupts to the Z80 when pressing RESTORE or from CIA #2. Furthermore, the I/O port registers at addresses 0 and 1 are 8502 specific so you can't change anything related to those bits when the Z80 is active, i.e. you can't decide if the CPU and the VIC should see color RAM from bank 0 or 1 (a little known feature), you can't use a Datassette, you can't detect if CAPS LOCK has been pressed and you can't switch out the character ROM shadow that by default exists in all 4 VIC banks on the C128. (On the C64 you always have character ROM shadow in two of the VIC banks and never in the other two but with the C128 with the 8502 active you can select if you want to have character ROM shadow in all four banks or in none of the banks.)

- Commented Z80 example source code for Power Assembler / ZBUD written by me (Christian Johansson)(for example a VIC text scroller)
- .d64 file with the Z80 source code above (uncommented) + some other C128 programs I have written

80-column mode

In addition to the VIC chip, the Commodore 128 also has another graphics chip that the Commodore 64 does not have. The additional graphics chip is called 8563 and is often referred to as Video Device Controller (VDC).

The output from the VDC chip comes through the RGBI port at the rear of the Commodore 128. RGBI means Red Green Blue Intensity. Since all these four properties can be either 0 or 1, there are a total of 16 colours. Three of the 16 colours are different between the VIC chip and the VDC chip. The rest of the colours are the same. The RGBI port should preferrably be connected to a monitor that can handle RGBI signals. Some Commodore monitors can handle RGBI signals. All CGA monitors can also do this. The VDC chip is actually a Commodore implementation of a CGA chip. CGA was a graphics standard that was used by many PCs at the time when Commodore 128 was launched. (CGA was later followed by EGA, VGA and SVGA). It is possible to connect a cable between the RGBI port of the C128 and the composite video input of a monitor or a TV set but this gives a black & white picture only (with a few different shades of grey). I also have an RGBI -> SCART cable but I haven't tested it so I can't say if it gives colours. In contrast to the VIC chip, it is not possible to connect the output of the VDC chip to a TV set using an RF cable.

The VDC chip gives 80 columns on the screen in contrast to the VIC chip that only gives 40 columns. This means that the horizontal resolution is doubled from 320 pixels to 640 pixels when using the VDC chip instead of the VIC chip. The VDC chip is mostly intended for characters and in contrast to the VIC chip the VDC chip only has very limited graphics modes. Sprites (movable graphics objects) do not exist at all when using the VDC chip. This makes the VIC chip much better than the VDC chip for games while the VDC chip is excellent for word processing and other applications using characters. With the VIC chip it is possible to have 128 different characters + their reverse counterparts on the screen at the same time. With the VDC chip, if you reverse a character, it does not mean that the screen code is increased by 128 as for the VIC chip. It simply means that background/foreground is reversed for all pixels of the character. This makes it possible to have 256 different characters + their reverse counterparts on the screen at the same time. As if this wasn't enough, it is also possible to specify for each character if it should be taken from the character set with capitals/graphics or from the character set with small letters/capitals (for the VIC chip, the character set cannot be selected for each character individually but only for the whole screen). This gives a total of 512 different characters on the screen at the same time. For each character it can also be specified if it should be reverse, underlined or blinking (and of course which colour it should have out of 16 colours where 13 of the colours are the same as for the VIC chip while 3 are VDC specific).

In contrast to the VIC registers, the VDC registers are not a part of the Commodore 128 memory map. Instead, all accesses to the 38 VDC registers are made through

two C128 registers; the address register at $D600 and the data register at $D601. A read or write access to a VDC register is made by writing the VDC register number (0-37) to the C128 register at address $D600 and then wait for bit 7 of the C128 register at address $D600 to become equal to 1. When it has become equal to 1, a read from or write to the C128 data register at address $D601 can be performed.

The VDC chip in Commodore 128 DCR (the metal case model that was mainly sold in North America) has 64 kB of RAM memory while the VDC chip in all other Commodore 128 models by default has 16 kB of memory (expandable to 64 kB). Commodore 128 computers where the VDC chip has 64 kB of RAM memory have better graphics capabilities when the VDC chip is used. The VDC memory is not a part of the normal 128 kB of RAM memory that the Commodore 128 has. The VDC memory is used for the screen memory (which characters or graphics that are shown on the screen), for character or graphics attributes (for example colours) and for character definitions. The VDC memory is accessed through VDC registers. Simple access to the VDC memory is made by writing the VDC memory address to VDC registers 18 and 19 and then read the data from or write the data to VDC register 31. There are also more advanced operations for copying blocks of data. Memory addresses are written to VDC registers with the most significant byte in the register with the lowest address in contrast to C128 registers where the least significant byte of memory addresses is written to the register with the lowest address.

In contrast to the VIC chip, there is no raster interrupt for the VDC chip. The C128 has exactly the same interrupt sources (from the VIC and CIA chips) as the C64.

It is possible to use the VDC chip in C64 mode on a C128 but this is not a trivial task since the Kernal in the C64 does not support using the VDC chip and routines from the C128 Kernal would therefore have to be copied into the C64 RAM memory and be modified appropriately. The terminal program Novaterm is an example of a program that uses the VDC chip in C64 mode. A much easier thing to do is to use the VDC RAM memory as extra memory in C64 mode. By doing this, you will have 80 kB or 128 kB of RAM memory available in C64 mode instead of 64 kB (although the VDC RAM memory is more difficult and slower to access than the normal C64 RAM memory).

A bug that exists in older revisions of the VDC chip is that sometimes data can be lost on a read or a write. This is due to a synchronization problem. Another bug that exists is that the block copy function sometimes copies one byte too little. There is a workaround in the C128 Kernal for the block copy bug. I have received information from a person who has worked a lot with the VDC chip that out of five C128s he knows of only one has a VDC chip without bugs.

BASIC incompatibilities

Commodore claimed that a BASIC program written for BASIC 2.0 (Commodore 64) should also work under BASIC 7.0 (Commodore 128 in Commodore 128 mode) provided that the program does not POKE to or PEEK from memory locations that have different meanings on C64 and C128. However, I have found this not to be true. I think that the incompatibilities are important enough to be described on this page even though the page is mainly intended for assembly programming.

One incompatibility problem occurs when having a PRINT statement that writes the last character on the last column of a row. Under BASIC 7.0, the next PRINT statement then begins on the next row but under BASIC 2.0, the next PRINT statement begins on the row after the next row. This problem can be overcome by ending the PRINT statement with a ";". Then, the next PRINT statement begins on the next row both under BASIC 7.0 and under BASIC 2.0. However, there is also another incompatibility problem that adding a ";" does not solve. The problem is that when having a PRINT statement that writes the last character on the last column of a row, BASIC 7.0 moves all the following rows one row down while BASIC 2.0 does not affect the following rows. The only way of solving this problem is probably that when the screen needs to be updated, all the rows below the first row that is rewritten must also be rewritten. Then, it will work both under BASIC 2.0 and under BASIC 7.0. Because of the just mentioned problems, most BASIC programs I have written for BASIC 2.0 look wrong on the screen when they are run under BASIC 7.0.

As mentioned under Sprites, the BASIC part of the IRQ routine on the Commodore 128 reads from shadow registers and writes to actual VIC and SID registers. This means that BASIC 2.0 programs that use sprites and sound will not work under BASIC 7.0 (since they write directly to VIC and SID registers) unless the BASIC part of the IRQ routine is disabled. The disabling can be done by setting bit 0 of the system variable at address $0A04 equal to 0. Note that on a Commodore 64, address $0A04 is a part of the BASIC text area so writing to this memory location would destroy the BASIC program. Therefore, it needs to be checked that the program is running on a Commodore 128 in Commodore 128 mode before writing to this memory location.

As mentioned under Graphics, the raster-interrupt driven screen editor on the Commodore 128 reads from shadow registers and writes to actual registers. This means that BASIC 2.0 programs that write directly to registers $D011, $D016 or $D018 in order to enable/disable graphics modes or to set pointers to screen memory, beginning of character data or bit map data will not work under BASIC 7.0 unless the screen editor is disabled. The screen editor can be disabled by writing $FF to the system variable at address $00D8. Since the memory location at address $00D8 has another use on the Commodore 64 than on the Commodore 128, it first needs to be checked that the program is running on a Commodore 128 in Commodore 128 mode before writing to this memory location.

To detect if a program is running on a Commodore 128 or on a Commodore 64, the CPU vectors can be read and compared. For example, the low byte of the BREAK/IRQ vector at address $FFFE contains $48 if running on a Commodore 64 (or on a Commodore 128 in Commodore 64 mode) and $17 if running on a Commodore 128 in Commodore 128 mode.

Incompatibilities between real C64 and C128 in C64 mode

Almost all programs that have been implemented for the C64 also work on the C128 in C64 mode. This section describes why a few programs do not work.

A main cause of a C64 program not working on the C128 in C64 mode is that the program writes to $D030. On the C64, this is an unused memory address but on the C128 in C64 mode it is used for clock frequency setting (1 or 2 MHz) and it also contains a test bit for the VIC-IIe chip. Therefore, writing certain values to $D030 on the C128 in C64 mode makes it appear as if the computer has crashed since the screen will go blank or will be filled with garbage. The other VIC address that is unused on the C64 but used on the C128 is $D02F, which is used for extra columns in the keyboard matrix. However, there is no harm in writing to this register on a C128 in C64 mode. Actually, it is a good memory location to use for detecting if the program is running on a real C64 or on a C128 in C64 mode. On a real C64, this memory location always contains the value $FF no matter what is written to it but on a C128 running in C64 mode, the value of the register can be changed.

Another cause of C64 programs not working on the C128 in C64 mode is that the SID register mirror images are missing on the C128 in C64 mode. At $D500, MMU registers exist on the C128 (invisible in C64 mode) and at $D600 the VDC port registers are placed.

On the C128, both in C128 and in C64 mode, bit 6 in register $01 is used to tell whether the CAPS LOCK/ASCII DIN/ASCII CC key is pressed or not. This may cause certain C64 programs that do not mask the bits in register $01 properly to fail.

I have received information about that there are compatibility problems with some C64 game cartridges on the C128. This especially concerns cartridges that hack in /EXROM on reset for some cycles. As I have understood it, the problem is that on the C128 it takes more cycles until the system reaches C64 mode and the Kernal routine that checks for the signature CBM80 at $8000 after a reset (this signature indicates that a C64 cartridge is present).

Hämtad från "http://commodore64.se/wiki/index.php/Commodore_128_assembly_programming"

BANNERS