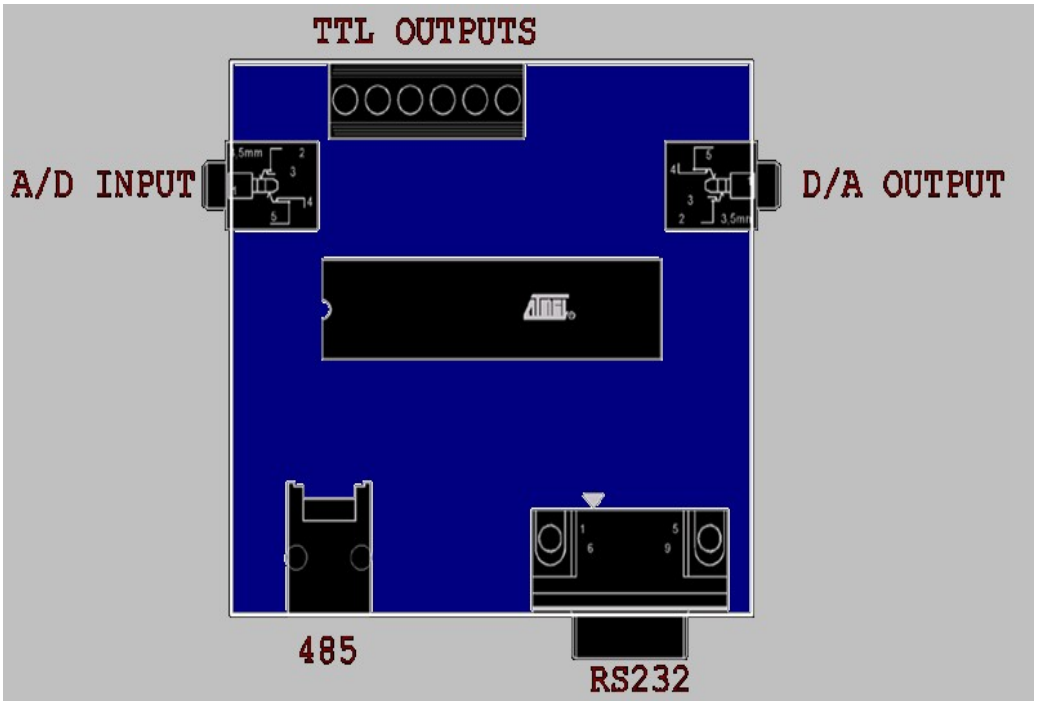

HOW TO USE THE CPC BOOSTER+

The CPC booster+ is very easy to use, even through BASIC. In this document I will explain how to use all the functions of the card by analyzing the memory map. I will use both Assembly and Basic, but in most cases you will have to use only assembly to use the speedy characteristics (you can't have 230400 baud serial communication in Basic!). The only thing you need is simple IN and OUT commands to give orders to the board. All the addresses are 16bit, but the high byte is always &FF for the CPC booster.

When you switch on your CPC, after one second the led of the card will be turned on. This means that the card is working. If during the operation the led flashes it means that the microcontroller is resetting itself because the power supply is insufficient. This may happen if you have an external drive connected which takes power from the CPC and not from an external power supply or if you have too many peripherals connected. In this case, send me an e-mail and we'll see what we can do to avoid this problem. Be sure that you have connected the cable in the right way otherwise the led won't flash at all.

The CPC Booster+ is an open source project. Since most of you know Z80 assembly, then it will be easy for you to write your own routines in AVR assembly, which is very similar to the Z80. Right now, there is a LOT of free space in the bios of the booster to fill it with anything you want. Imagine that the booster has a faster processor not only because of the crystal frequency of the 11.05292 MHz, but AVR is also a RISC processor, almost every command is executed in 1 machine cycle.

If you are willing to add your own routines, then don't hesitate to contact me in order to give you all the software and details you need to write your own stuff. But I would advice you to send the changed source code back to me in order to spread it and keep one version of the bios for everyone. Don't forget that an update of the bios is possible through the CPC.

If you also have trouble on using the booster, even after reading the manual, then contact me and perhaps we can make a forum that we could discuss about it. I can tell you that the booster's capabilities can be a very long subject to analyze.

The CPC Booster hardware includes the following things:

-One RS232/485 serial port.

-Two Analog to digital converters (8 Bit) with rec level
-Two PWM channels, used as digital to analog converters
-One 5bit TTL in/out port
-512 Bytes EEPROM
-256 Bytes RAM buffer
-Keyboard scanning functions

WARNING: Before you use any of the fast routines, in most cases you have to disable the interrupts first:

```
      LD HL,&C9FB
      LD (&38),HL

      Or use DI
```

Let’s start describing all the functions:

```
;*****************************************************************
;MEMORY MAP OF THE CPC BOOSTER+

;00         IN          10101010          TEST BYTE #1
            OUT         00-$FF            RESET BOOSTER
;01         IN          01010101          TEST BYTE #2
            OUT         00-$FF            RESET BOOSTER

;02         OUT         00-$FF            PWM CHANNEL 1
;03         OUT         00-$FF            PWM CHANNEL 2

;04         IN/OUT      00-$FF            UBRR/BAUD RATE
;05         IN/OUT      00-$FF            UDR READ/WRITE
;06         IN/OUT      00-$FF            UART REG 1
;07         IN/OUT      00-$FF            UART REG 2
;08         OUT         00-$FF            UART TX/AUTO POLLING
;09         IN          00/$FF            UART WAIT UDR CHARACTER
;0A         IN/OUT      00-$FF            UART READ TIME OUT*50ms
;0B         IN/OUT      000XXXXX          UART REG 3

;0C         IN/OUT      00-$01            EEPROM ADDRESS HIGH
;0D         IN/OUT      00-$FF            EEPROM ADDRESS LOW
;0E         IN/OUT      00-$FF            EEPROM READ/WRITE

;0F         IN/OUT      00000XXX          ADC SAMPLING FREQUENCY
;10         IN/OUT      00-$01            ADC CHANNEL SELECTION
;11         IN          00-$FF            READ ADC VALUE

;12         IN/OUT      00-$FF            KEYBOARD READ

;13         OUT         00-$7F            PAGE WRITE FOR UPDATE
;14         OUT         00-$FF            DATA FOR UPDATE BUFFER
;15         IN/OUT      00-$7F            ADDRESS OF BUFFER FOR BIOS UPDATE

;16         IN/OUT      00-$7F            ROM PAGE NUMBER
;17         IN/OUT      00-$7F            ADDRESS OF PAGE
;18         IN          00-$FF            READ ROM DATA (PAGE MODE)

;19         IN/OUT      00-$3F            ROM ADDRESS HIGH
;1A         IN/OUT      00-$FF            ROM ADRESS LOW
;1B         IN          00-$FF            READ ROM DATA (ADDRESSING MODE)

;1C         IN          00-$FF            AVAILABLE CHARACTER IN UART BUFFER
```

	OUT	00-&FF	RESET UART BUFFER
;1D	IN	00-$FF	READ CHARACTER FROM BUFFER
;1E	IN/OUT	00-$1F	5 BITS PORT DIRECTION SETTING
;1F	IN/OUT	00-$1F	5 BITS PORT LATCH (OUTPUT)
;20	IN	00-$1F	5 BITS PORT INPUT
;21	OUT	00-$FF	MULTIPLIER 1
;22	OUT	00-$FF	MULTIPLIER 2
;23	IN	00-$FF	RESULT HIGH BYTE
;24	IN	00-$FF	RESULT LOW BYTE
;25	IN	00-$FF	READ VERSION
	OUT	00-$FF	RESET TEXT ADDRESS
;26	OUT	00-$FF	PWM BUFFERED STEREO
;27	OUT	00-$FF	PWM MONO TO BOTH CHANNELS
;28	IN/OUT	00-$FF	RAM BUFFER ADDRESS
;29	IN/OUT	00-$FF	RAM BUFFER DATA
;2A	IN/OUT	00-$FF	RAM BUFFER DATA POST INCREMENT

ADDRESS:&FF00 + &FF01 IN/OUT TEST BYTES / RESET

Those two addresses are used to test the board, mainly to check if the connector's cable is working. If you type in BASIC

```
?INP(&FF00),INP(&FF01)
```

and you get the results 170 , 85 then the board is working. Address 00 always reads 170 and address 01 always reads 85. If you get any other values, then there must be a problem with your cable. Try moving the cable a little bit till you get the correct values. The CPC connector is a problematic one ☺

If you make an OUT any value to those addresses, then the CPC booster makes a Reset.

OUT &FF00,n or OUT &FF01,n (n = any value from 0 to 255)

ADDRESS:&FF02 + &FF03 IN/OUT PWM CHANNELS

The board has two 8BIT PWM channels. PWM stands for Pulse Width Modulation, which means that it is an output that gives you pulses which we can alter their width. To alter the width, we send 8bit values to those addresses. To send a value to channel 1 for example, you type in BASIC

OUT &FF02,X where X is the value we want to send

In assembly we type

```
LD BC,&FF02
LD A,X
OUT (C),A
```

It's the same thing for channel two: instead of &FF02, we use &FF03. If you make an in on any of those two channels, you can read the last value you sent. The output needs a pre-amplifier, or just a good amplifier. Due to the low-pass filter, which is used to turn the pulses into DC signals (the Digital to analog converter) , you will get a more bass sound. No need for digiblaster or soundplayer if you have a CPC Booster+, you can play stereo samples and the quality is good.

ADDRESS:&FF26 IN/OUT PWM BUFFERED STEREO

In order to have values played on both channels at the same time, you can use this address. First you send the value for PWM channel 1 which is buffered and when you send the value for PWM channel 2, both values are transferred to the output at the same time. If you make an IN, you clear the buffer and the routine is waiting again for a value for PWM channel 1. You can clear the buffer at first, you don't have to clear it after every two values you send.

```
      LD BC,&FF26
      IN A,(C)          ;Clear the buffer
LOOP: LD A,X            ;Value for channel 1
      OUT (C),A         ;Store the value to the buffer
      LD A,Y            ;Value for channel 2
      OUT (C),A         ;Now send both values to PWM channels
      JR LOOP
```

ADDRESS:&FF27 IN/OUT PWM VALUE TO BOTH CHANNELS

If you want to play mono samples at both channels, to have one value played at the same time to the two PWM channels, you can use this address.

```
      LD BC,&FF27
      LD A,X
      OUT (C),A         ;Value is transfered to both channels PWM 1 & 2
```

THE USART – SERIAL COMMUNICATION

The addresses from &FF04 to &FF0B are used for the UART, which means Universal synchronous/asynchronous receiver-transmitter. Or just RS232. I suggest you to use assembly though it's possible to use Basic at low Baud rates. An important thing in high speed communication is to disable the interrupts first. We can control and set up the RS232 using three registers of the CPC Booster+: UART REG1, UART REG2 and UART REG3.

The CPC Booster+ has also an RS485 network. You can transmit to the RS485 and the RS232 at the same time but you can only read data from one of them. There's a switch on the board which selects from which port to read data.

ADDRESS:&FF04 IN/OUT UBRR (Baud rate)

The address &FF04 is used to select the baud rate of the UART. It's a value between 0-255 and it's calculated like this:

```
UBRR= (( FREQUENCY / BAUDRATE) / 16 ) - 1
BAUDRATE= FREQUENCY / ((UBRR+1)*16)
```

In our case:
UBRR= ((11059200 / BAUDRATE) / 16) - 1
There are two modes to select the baud rate. The normal and the double speed (U2X) can be selected in the UART REG3 which will be described later.

UBRR	U2X=0 BAUDRATE	U2X=1 BAUDRATE
4800	143	X
9600	71	143
14400	47	95
19200	35	71

28800	23	47
38400	17	35
57600	11	23
115200	5	11
230400	2	5
345600	1	3
691200	0	1
1382400	X	0

If we want to select 57600, then we type in BASIC

```
OUT &FF04,11
```

And in assembly

```
LD A,11
LD BC,&FF04
OUT (C) ,A
```

We have the ability to read also the UBRR value we've selected

```
?INP(&FF04) in BASIC
```

and in assembly:

```
LD BC,&FF04
IN A,(C)
```

ADDRESS:&FF05 IN/OUT UDR READ/WRITE

This address has actually two separate functions, one for IN and one for OUT. When we use the IN command, we read the RX input of the UART and when we make an OUT, we transmit a value to the UART. But in order to use the UART properly, we will have to examine the flags first.

ADDRESS:&FF06 IN/OUT UART REGISTER 1

RXC
TXC
UDRE
FE
DOR
PE
RXB8
TXB8

This register contains 6 flags in order to control the UART. To read them we use in Basic

```
?INP(&FF06)
```

and in assembly

```
LD BC,&FF06
IN A,(C)
```

BIT 7 – RXC: UART RECEIVE COMPLETE

This bit is set when the UART has received a character. So before we use the address &FF05 to read a value, we have to check first this bit to see if a character was received. This bit is cleared by reading the UDR (&FF05).

BIT 6 – TXC: UART TRANSMIT COMPLETE
This bit is set when the entire character was transmitted, including the stop bit. This is used mainly for half duplex communication, where you have to know when your character has been transmitted before you send the next one or to enter receive mode. This bit is cleared by writing a logical one to the bit.

BIT 5 – UDRE: UART DATA REGISTER EMPTY
This bit is set (one) when a character written to UDR (&FF05) is transferred to the transmit shift register of the microcontroller and the UDR is empty. When this bit is set it means that we can send a new character to UDR (&FF05). This bit is cleared when we send a character to UDR.

BIT 4 – FE: FRAMING ERROR
This bit is set if a framing error condition is detected, i.e. when the stop bit of an incoming character is zero. The FE bit is cleared when the stop bit of received data is one.

BIT 3 – OR: OverRun
This bit is set if an overrun condition is detected, i.e. when a character already present in the UDR register is not read before the next character has been shifted into the receiver shift register. The OR bit is buffered , which means that it will be set once the valid data still in UDRE is read. The OR bit is cleared when data is received and transferred to UDR.

BIT 2 – PE: Parity error
This bit is set if the next character of the UART had a parity error when received and the the parity checking was enabled at that point. This bit is valid until the UDR (&FF05) is read. Always set this bit to zero when writing to UART REG1.

BIT 1 – RXB8: Receive data bit 8
RXB8 is the ninth data bit of the received character when operating with serial frames with nine data bits. Must be read before reading the low bits from UDR.

BIT 0 – TXB8: Transmit data bit 8
TXB8 is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. Must be written before writing the low bits to UDR.

OK, if those bits seem like chinese, don’t worry. Here are some routines to make things more clear:

BASIC – Receiving data

```
10 A=INP(&FF06)         ;READ THE FLAGS
20 A=A AND 128          ;CHECK THE RXC FLAG
30 IF A=0 THEN 10       ;IF RXC IS NOT SET THEN GOTO 10
40 A=INP(&FF05)         ;READ THE CHARACTER (RXC IS NOW CLEARED)
50 PRINT A
60 GOTO 10
```

ASSEMBLY – Receiving data

```
            LD BC,&FF06
RX_LOOP:    IN A,(C)        ;Read the flags
```

```
            AND A,%10000000 ;Check if RXC bit is set
            JR Z,RX_LOOP    ;If zero, goto RX_LOOP
            DEC C           ;Select &FF05
            IN A,(C)        ;Read the received character and clear the RXC
```

BASIC – Sending a character / HALF DUPLEX communication

```
10 A=X                  ;X is an 8bit character we want to transmit
20 OUT &FF05,A          ;Send character
30 A=INP(&FF06)         ;Read the flags
40 A=A AND 64           ;Leave only the TXC flag
50 IF A=0 THEN 30       ;Wait till the TXC flag is SET. The flag will be set when
the entire character is transmitted
60 A=64                 ;You can skip this since A is already 64
70 OUT &FF06,A          ;Set the TXC bit to clear it (strange, isn't it?)
80 GOTO 10
```

ASSEMBLY- Sending a character / HALF DUPLEX communication

```
            LD BC,&FF05
            LD A,X              ;X= data we want to transmit
            OUT(C),A            ;Send the character to UDR
            LD BC,&FF06
TXC_LOOP:   IN A,(C)            ;Read the flags
            ANDI A,%01000000    ;Check the TXC
            JR Z,TXC_LOOP       ;If cleared, goto TXC_LOOP
            LD A,64             ;You can skip this
            OUT (C),A           ;Clear the TXC by writing a logic one to it.
```

BASIC – Sending a character / FULL DUPLEX communication

```
10 A=INP(&FF06)         ;Read the flags
20 A=A AND 32           ;Check the UDRE bit
30 IF A=0 THEN 10       ;If UDRE is cleared, then the UART is not ready to send a
new character
40 A=X                  ;X= Data we want to transmit
50 OUT (&FF05),A        ;Transmit data
```

ASSEMBLY – Sending a character / FULL DUPLEX communication

```
            LD BC,&FF06
UDRE_LOOP:  IN A,(C)             ;Read the flags
            AND A,32             ;Check the UDRE bit
            JR Z,UDRE_LOOP       ;If zero, goto UDRE_LOOP
            LD A,X               ;X= data we want to transmit
            LD BC,&FF05
            OUT(C),A             ;Transmit character
```

ADDRESS:&FF07 IN/OUT UART REG2

UMSEL
UPM1
UPM0
USBS
UCSZ2
UCSZ1
UCSZ0
UCPOL

This register contains settings for the UART.

BIT 7 – UMSEL: USART mode select

This bit selects between asynchronous (0) and synchronous (1) mode of operation.

BITS 6,5 – UPM1,UPM0: Parity Mode

These bits enable and set type of parity generation and check. If enabled, the transmitter will automatically generate and send parity of the transmitted data bits within each frame. The receiver will generate a parity value for the incoming data and compare it to the UPM setting. If a mismatch is detected, the PE flag in UART REG1 will be set

UPM1	UPM0	Parity mode
0	0	Disabled
1	0	Even parity
1	1	Odd parity

BIT 4 – USBS: Stop bit select

This bit selects the number of Stop bits to be inserted by the transmitter. The receiver ignores this setting.

USBS=0 1 stop bit
USBS=1 2 stop bits

BIT 3,2,1 – UCSZ2,UCSZ1,UCSZ0: Character size

The UCSZ2:1:0 bits set the number of data bits (character size) in a frame the receiver and transmitter use.

UCSZ2	UCSZ1	UCSZ0	Character size
0	0	0	5 bit
0	0	1	6 bit
0	1	0	7 bit
0	1	1	8 bit
1	1	1	9 bit

BIT 0 – UCPOL: Clock polarity

This bit is used for synchronous mode only. Write this bit to zero when asynchronous mode is used. The UCPOL bit sets the relationship between data output change and data input sample, and the synchronous clock (XCK).

	Transmitted data changed (TX pin)	Received data sampled (RX pin)
UCPOL=0	Rising XCK edge	Falling XCK edge
UCPOL=1	Falling XCK edge	Rising XCK edge

ADDRESS:&FF08 OUT TX – AUTO POLLING

If you want to transmit a character in a full duplex or a half duplex communication then you just send the character to the address &FF08 and the flags are automatically checked by the program of the microcontroller!

```
BASIC:      A=X
            OUT &FF08,A

ASSEMBLY:   LD BC,&FF08
            LD A,X
            OUT(C),A
```

To select the type of communication you want to have (full duplex, half duplex or 485 halfduplex), you can use UART REG3.

ADDRESS:&FF09 IN RX – WAIT UDR CHARACTER
ADDRESS:&FF0A IN/OUT TIME OUT VALUE * 50msec

The address &FF09 uses a time-out function. When you make an IN from this address, if a character is received, you'll read the value 255, if no character appears after a time-out then you will read the value 0. To set the time-out, we use the address &FF0A. Examples in BASIC and assembly:

BASIC – Receiving a character

```
10 OUT &FF0A,10          ;Set the time out at 10*50= 500 msec
20 A=INP(&FF09)          ;Check if there is an available character in UDR
30 IF A=0 THEN 50        ;If after 500msec no character appears then we
                         ;get the result 0
40 A=INP(&FF05):END      ;Else, we read the received character
50 PRINT "NO CHARACTER"
```

ASSEMBLY- Receiving a character

```
            LD A,10
            LD BC,&FF0A
            OUT(C),A          ;Set the time out at 10*50= 500msec
            LD BC,&FF09
RX_LOOP:    IN A,(C)          ;Check if there is an available character in UDR
            CP A,0            ;If A=0 then goto RX_LOOP
            JR Z,RX_LOOP
            LD BC,&FF05
            IN A,(C)          ;Read the received character
```

As I said above, when you make an IN from address &FF09, if no character is received, you'll get the value 0 after the time out, in our example the 500msec. But as soon as a character is received, you'll get immediately the value 255, you won't have to wait the 500msec to pass.

ADDRESS:&FF0B IN/OUT UART REG3

UART BUFFER ON/OFF
U2X
485 AUTO POLLING
FULL/HALF DUPLEX
MASTER/SLAVE

Only the 5 low bits are used in this register. Bits 7,6 and 5 are always read as zero.

BIT 4 – UART BUFFER ON/OFF: Enables/disables the UART buffer.
The CPC booster has a buffer of 255 bytes for the UART which improves the communication. It can be enabled by writing 1 to this bit. More details about the buffer in the following addresses.

BIT 3 – U2X: Double UART speed
This bit selects between normal and double speed for the UART. To calculate UBRR when the double speed is selected you use this:
UBRR= ((FREQUENCY / BAUDRATE) / 8) – 1
BAUDRATE= FREQUENCY / ((UBRR+1)*8)

BIT 2 – 485 AUTO POLLING
If BIT 1 of UART REG3 is set (half duplex selected) then the Master/Slave pin of the 485 is automatically driven by the CPC Booster when you use the

routine of TX AUTO POLLING (&FF08). If BIT 1 is reset (full duplex selected), then this bit has no effect.

BIT 1 – FULL/HALF DUPLEX

With this bit you can select the type of communication to use with the TX AUTO POLLING (&FF08) routine.
0=FULL DUPLEX
1=HALF DUPLEX

BIT 0 – MASTER/SLAVE

This bit directly drives the pin of TX/RX enable of the 485. Whenever you want to transmit something to the 485 you have to set this bit and after the transmission is over, you have to reset it. By setting BIT 1 and BIT 2 of the UART REG3, this pin is driven automatically by the CPC booster when you use the TX AUTO POLLING routine.

Except for the 485, this bit is also connected to the RS232 port as the Carrier Detect signal.

UART BUFFERING

While I was trying to make my terminal program, I figured out that our CPC is kind of slow for terminal emulation. Imagine what a terminal program does: Checks for incoming characters, prints them, scans the keyboard, prints and transmits the pressed keys. The CPC was losing bytes even at very low speed. At first I thought of installing a buffer on the CPC. I did that and there was a huge improvement but still every now and then , some bytes were missing. So I thought about installing a buffer inside the CPC booster. And it really works! The communication is perfect even at 230400!

The UART buffer is 255 bytes long. It can be enabled or disabled. If the buffer is enabled then you shouldn't use &FF05 or &FF06 to read incoming characters cause you will mess with the buffer. If the buffer is disabled you use the old routines without changing anything. Enabling the buffer affects only the incoming characters, the transmitting methods (half duplex/full duplex) remain the same.

ADDRESS:&FF1C IN/OUT NUMBER OF AVAILABLE BYTES IN BUFFER / RESET BUFFER
ADDRESS:&FF1D IN READ DATA FROM BUFFER

If the buffer is enabled, then the address &FF1C contains the number of available characters in the buffer when you make an IN. Making an OUT any value at this address, resets the buffer.

To enable/disable the buffer, you use BIT 4 of UART REG3.

When you make an IN from the address &FF1D, then you read the incoming data from the buffer, and the number of available bytes in the buffer decreases. When characters are received, then the number of bytes in the buffer increases.

An example in BASIC

```
10 OUT &FF0B,16              ;Enable the buffer
20 OUT &FF1C,0               ;Reset the buffer
30 IF INP(&FF1C)=0 THEN 30   ;Wait for incoming characters in the buffer
40 ?INP(&FF1D)               ;Print data from the buffer
50 GOTO 30
```

An example in Assembly (don't worry, I know that the code is not optimized :))

```
      LD A,16
      LD BC,&FF0B
      OUT(C),A               ;Enable the buffer
```

```
      LD A,1
      LD BC,&FF1C
      OUT(C),A                ;Reset the buffer

LOOP: LD BC,&FF1C
      IN A,(C)
      CP A,0                ;Loop until a character appears
      JR Z,LOOP
      LD BC,&FF1D
      IN A,(C)                ;Read data from buffer
      CALL &BB5A              ;Print data
      JR LOOP
```

512 BYTES EEPROM

Since the microcontroller had the EEPROM, I thought that I should give the ability for the CPC to access it. I know it's not much but incase you want to save some settings, it's quite good. Anyway, the addresses are from 0 till 511 (0-&1FF in HEX). **Address 0 shouldn't be used because it's reserved for the bios protection routine.**

ADDRESS:&FF0C IN/OUT EEPROM ADDRESS HIGH BYTE
ADDRESS:&FF0D IN/OUT EEPROM ADDRESS LOW BYTE

The high byte can only be 0 or 1, a value bigger than 1 will be automatically changed to 1

ADDRESS:&FF0E IN/OUT EEPROM READ/WRITE

This accesses the byte of the address of the EEPROM we've selected with the addresses &FF0C and &FF0D. Here are some examples in BASIC

```
10 OUT &FF0C,0
20 OUT &FF0D,5          ;Select address 5
30 OUT &FF0E,25         ;Store the value 25 at the address 5
40 OUT &FF0C,&1
50 OUT &FF0D,&4E        ;Select address &14E (334 in decimal)
60 A=INP(&FF0E)         ;Read the contents of the address &14E
70 PRINT A
```

THE ANALOG TO DIGITAL CONVERTER

The CPC booster gives you the ability to make 8bit samples on the CPC. Ofcourse, there were some samplers in the past for the CPC, like the Music Machine, but this time it's easier than ever! The sampler can be used only through assembly because in Basic the sampling rate is worse than an 1 bit sample!

ADDRESS:&FF0F IN/OUT ADC SAMPLING FREQUENCY

With this address, we can select the sampling rate. We don't have to make complicated routines on the CPC, the baud rate can be set directly on the CPC booster, so we just make a simple IN to get the data and all the timing is done automatically. There are 8 possible values to select frequency, value 0 and 1 are the same:

VALUE	DIVISION FACTOR	SAMPLING FREQUENCY
0	2	5529 KHz
1	2	5529 KHz
2	4	2764 KHz
3	8	1382 KHz
4	16	691 KHz
5	32	345 KHz
6	64	172 KHz
7	128	86 KHz

As you understand, if the frequency is high, you have good quality but it takes a lot of memory and you can't make long samples. In the worst quality, 86 KHz, you can create around 14 sec long sample, which takes 64 KB.

```
ASSEMBLY:       LD A,3                  ;Select 1382 KHz sampling freq.
                LD BC,&FF0F
                OUT(C),A
```

ADDRESS:&FF10 IN/OUT ADC CHANNEL SELECTION

With this address you select from which channel you will get the data. There are two channels, so we have the possibility to make stereo samples, or to record a stereo sample from a music CD and turn it into mono. To select channel 1, we send the value 0, to select channel 2 we send value 1. Any value between 2 and 255 selects channel 1 again (I've just noticed that!).

```
ASSEMBLY:       LD A,0              ;Select channel 1
                LD BC,&FF10
                OUT(C),A
```

ADDRESS:&FF11 IN READ ADC VALUE

To read the data from the A/D convertor, we make an IN from this address. Could it be simplier? Here's a routine which stores a sample with max width 10KB

```
            LD HL,16384         ;Address to store the sample
            LD DE,10000         ;Byte counter (10KB long sample)
            LD BC,&FF11         ;Address for reading data
SAMPLE:     IN A,(C)            ;Read byte from the convertor
            LD(HL),A            ;Store byte in the address that HL points
            INC HL              ;Increase sample address to store next byte
            DEC DE              ;Decrease counter
            LD A,D
            OR E
            JR NZ,SAMPLE        ;Check if sample is 10KB long.
```

During the recording of a sample, you can adjust the incoming record volume with the potensiometer located on the top of the CPC booster. Use a screwdriver and adjust it till you get a clear sound. A way to do this adjustment is to make an in with &FF11 and send the value directly to the PWM.

ADDRESS:&FF12 IN/OUT KEYBOARD SCANNING FUNCTIONS

The functions on the address &FF12 are combined with a routine on the CPC. What this routine does is by having the 10 bytes of the CPC keyboard scan, can return to you the pressed key. There is a table which will help you find out the exact key pressed. This routine can only be used in an editor, when you can only press one key at a time (combined with shift).

First of all, let's see the routine on the CPC. What this routine does is making the whole keyboard scan and sending each line's byte to the CPC Booster.

```
            LD BC,&FF12:IN A,(C)    ;Making a false read in the beginning of the
program just to reset the routine
            ... MAIN PROGRAM ...
            CALL KEYSCAN            ;Call routine to scan the CPC keyboard
            LD BC,&FF12
            IN A,(C)                ;Get pressed key
            ... JUMP TO MAIN PROGRAM ...
KEYSCAN:    LD BC,&F40E:OUT(C),C
            LD BC,&F6C0:OUT(C),C
            DB &ED,&71
            LD BC,&F792:OUT(C),C
            LD BC,&F640:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A       ;
Read the byte from the scanned line
            LD BC,&F641:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A       ;
and send it to the CPC booster
            LD BC,&F642:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A
            LD BC,&F643:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A
            LD BC,&F644:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A
            LD BC,&F645:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A
            LD BC,&F646:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A
            LD BC,&F647:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A
            LD BC,&F648:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A
            LD BC,&F649:OUT(C),C:LD B,&F4:IN A,(C):LD BC,&FF12:OUT(C),A
            LD BC,&F782:OUT(C),C:LD BC,&F600:OUT(C),C:RET
```

The routine in the CPC booster, after receiving 10 bytes, it will automatically find the pressed key and will give you its value when you will make an IN from the address &FF12. If we send 9 bytes or any value less than 10, then we will read 0. You have to send 10 bytes to get a correct response. Here’s the table with the returned values:

00=No key	128=No key
01=a	129=A
02=b	130=B
03=c	131=C
04=d	132=D
05=e	133=E
06=f	134=F
07=g	135=G
08=h	136=H
09=i	137=I
10=j	138=J
11=k	139=K
12=l	140=L
13=m	141=M
14=n	142=N
15=o	143=O
16=p	144=P
17=q	145=Q
18=r	146=R
19=s	147=S
20=t	148=T
21=u	149=U
22=v	150=V
23=w	151=W
24=x	152=X
25=y	153=Y
26=z	154=Z

```
27=1                155=!
28=2                156=”
29=3                157=#
30=4                158=$
31=5                159=%
32=6                160=&
33=7                161=’
34=8                162=(
35=9                163=)
36=0                164=_
37=-                165==
38=^                166=EURO
39=@                167=|
40=[                168={
41=:                169=*
42=;                170=+
43=]                171=}
44=,                172=<
45=.                173=>
46=/                174=?
47=\                175=`
48=f0
49=f1
50=f2
51=f3
52=f4
53=f5
54=f6
55=f7
56=f8
57=f9
58=f.
59=cursor left
60=cursor right
61=cursor up
62=cursor down
63=CLR
64=DEL
65=RETURN
66=ENTER
67=SPACE
68=COPY
69=CAPS LOCK
70=CONTROL
71=TAB
72=ESC
```

As you can see, you can check bit7 to see if the shift key is pressed or not. To transform these values into ascii, you can use a 256 bytes page alligned table and a small routine:

```
LD BC,&FF12
IN A,(C)
LD L,A
LD H,High byte of the table’s address
LD A,(HL)
```

PROGRAM MEMORY READ/WRITE

The AVR mega16 has 16KB of program memory. The CPC booster gives you the ability to read and write this memory, that's why the update of the peripheral is possible. Right now, less than 5 KB of this memory is used for the bios of the CPC booster, so you have around 10KB for your own purposes. Ofcourse, you already know that you should be careful to which address you will store your data, because you may accidentally erase data. If this happens, then you should try to re-update the bios and if that fails, then you will have to send the microcontroller back to me to re-program it.

A few words about the memory of the microcontroller. It is divided into 128 pages of 128 bytes each. 128*128=16384 bytes. Every command of the AVR is 16bit, which means that it takes two bytes. There are two ways to use the program memory through the CPC Booster, the page mode and the address mode. In the page mode, you select the address using two registers, one contains the page number and the other the address inside the page. In the address mode, the memory is divided like a normal ram, there are 16384 addresses of 8 bit each. You use two registers, one for the high byte and one for the low byte.

You can use both ways when you want to read the program memory but you can only store data using the page mode.

ADDRESS:&FF13	**OUT**	**WRITE PAGE**
ADDRESS:&FF14	**OUT**	**STORE BYTES TO PAGE BUFFER**
ADDRESS:&FF15	**IN/OUT**	**ADDRESS OF THE PAGE BUFFER**

To write a page you have to fill the data into a buffer and then make an out at the address &FF13. To explain it better, here's a small basic program to store data to a page.

```
10 ADDRESS=16384              ;Source address data from the CPC
20 PAGE=80                    ;We will write the data to page 80 (0-127)
30 FOR BUFFER=0 TO 127        ;Storing 128 bytes
40 OUT &FF14,PEEK(ADDRESS)    ;Sending each byte to the CPC booster buffer. This
                               function auto increases the buffer's address.
50 ADDRESS=ADDRESS+1
55 NEXT BUFFER
60 OUT &FF13,PAGE             ;After filling the buffer, write the data to the
                               selected page.
```

The address &FF15, points at the next address of the buffer. If we make an OUT &FF14,X, then the contents of the &FF15 will be automatically increased. Ofcourse, when it reaches 128, it goes back to 0. You can use this address if you want to reset the buffer or to see how many bytes you've already stored inside the buffer. Kind of useless function but anyway.

WRITE BIOS PROTECTION

From bios V1.5A and on, to be able to write data to the program memory, you have to disable the write bios protect option. To disable write bios protect, you have to write the value &CC to the EEPROM address 0. Any other value, forbids the controller to write data to the program memory.

```
10 OUT &FF0C,0
20 OUT &FF0D,0
30 OUT &FF0E,&CC              ;DISABLE WRITE BIOS PROTECTION
```

ADDRESS:&FF16	**IN/OUT**	**PAGE NUMBER**
ADDRESS:&FF17	**IN/OUT**	**ADDRESS INSIDE PAGE**
ADDRESS:&FF18	**IN**	**READ DATA**

Those addresses are used to read data from the program memory using the page mode. Address &FF16 contains the page you want to read, address &FF17 contains the address inside the page (0-127) and using &FF18 after you set the previous addresses, you read the data.

EXAMPLE IN BASIC

```
10 PAGE=27
20 ADDRESS=0
30 OUT &FF16,PAGE
40 OUT &FF17,ADDRESS
50 ?INP(&FF18)
```

ADDRESS:&FF19	**IN/OUT**	**ADDRESS HIGH BYTE**
ADDRESS:&FF1A	**IN/OUT**	**ADDRESS LOW BYTE**
ADDRESS:&FF1B	**IN**	**READ DATA**

I don't think that you need further explanations... I remind you that the addresses are from 0 till 16383.

TTL INPUT/OUTPUT

This is a 5 bit port which can be used as input/output for TTL signals. I think that it's one of the most important characteristics of the CPC Booster because it gives you the ability to have a parallel port even if you only have 5 bits to control. People who like electronics and especially digital circuits already know how important this is.

The port is bi-directional with optional internal pull-ups.

ADDRESS:&FF1E	**IN/OUT**	**5 BIT PORT DATA DIRECTION (DDx)**
ADDRESS:&FF1F	**IN/OUT**	**5 BIT PORT DATA REGISTER (PORTx)**
ADDRESS:&FF20	**IN**	**5 BIT PORT INPUT (PINx)**

Those are the three addresses to control the port. Each one has 5 bits which control the 5 pins of the outport.

We can name the bits as DDx, PORTx and PINx. The DDx bit in the Data direction register selects the direction of the x pin (output or input). If the bit is zero, then the pin is an input, if it is set then the pin is an output.

If PORTx is set when the pin is configured as an input by the DDx bit, then the internal pull-up resistor is activated. To switch the pull-up resistor off, PORTx has to be zero or the pin must be configured as an output.

If PORTx is set when the pin is configured as an output pin, the port pin is driven high (5 Volts output). If PORTx is zero when the pin is configured as an output pin, then the port pin is driven low (0 Volts output).

Independent of the setting of the Data direction bit (DDx), the port pin can be read through the PINx register bit.

Some examples:

```
LD A,%11111
LD BC,&FF1E
OUT (C),A          ;All the pins are set as output pins
LD A,%10101
LD BC,&FF1F
OUT (C),A          ;Pins 1,3 and 5 are driven high and
                   ;pins 2 and 4 are driven low.

LD A,%11001
```

```
LD BC,&FF1E
OUT (C),A          ;Pins 1,4,5 are configured as outputs.
LD A,%10111
LD BC,&FF1F
OUT (C),A          ;Pins 1 and 5 are driven high. In pins 2 and 3,
                   ;the internal pull-ups are activated.
LD BC,&FF20
IN A,(C)           ;A has the state of each of the 5 pins.
```

A pull-up resistor is used when we set a pin as an input. Incase we have nothing connected to that pin externally, if we read the PINx register bit, we will get the value 1 because it's internally connected to the VCC through a resistor. A signal connected to the ground through a switch can drive this pin low. If you want to connect a button or a switch, one pin of the switch is connected to the ground and the other to the CPC Booster pin.

When the button is not pressed, you read "1" because of the internal pull-up. When the button is pressed, then the ground is connected to the pin and you read "0". I can't give you any more information about the TTL signals and the use of pull-ups because this is a manual for the CPC booster, not a lesson in electronics. But you can have a look at the PDF of the ATMega16, where the functions of the pins are described in a better way.

The pins are from left to right (TOP VIEW OF THE CPC BOOSTER):

GND, PIN1, PIN2, PIN3, PIN4, PIN5

MULTIPLICATION

ADDRESS:&FF21	**IN/OUT**	**MULTIPLIER 1**
ADDRESS:&FF22	**IN/OUT**	**MULTIPLIER 2**
ADDRESS:&FF23	**IN**	**RESULT HIGH BYTE**
ADDRESS:&FF24	**IN**	**RESULT LOW BYTE**

I think that this function is very easy. First you give a value for multiplier 1 and then you give a value for multiplier 2. Whenever you enter a value to multiplier 2, a multiplication between multiplier 1 and multiplier 2 is done and the result is stored in addresses &FF23 and &FF24. The result will remain intact as long as the next multiplication takes place.

This is a multiplication between two 8 bit numbers and the result will be a 16 bit number.

```
10 INPUT "MULTIPLIER 1";A
20 INPUT "MULTIPLIER 2";B
30 OUT &FF21,A
40 OUT &FF22,B
50 PRINT INP(&FF23)*256+INP(&FF24)
```

READ VERSION OF THE CPC BOOSTER+

ADDRESS:&FF25	**IN/OUT**	**READ VERSION / RESET TEXT POINTER**

You can read some info of the booster+ you have using this routine:

```
10 OUT &FF25,0          ;RESET TEXT POINTER
20 A=INP(&FF25)
30 IF A=0 THEN END
40 PRINT CHR$(A);
50 GOTO 20
```

256 BYTES RAM BUFFER

ADDRESS:&FF28	**IN/OUT**	**READ/WRITE RAM ADDRESS**
ADDRESS:&FF29	**IN/OUT**	**READ/WRITE RAM DATA**
ADDRESS:&FF2A	**IN/OUT**	**READ/WRITE RAM DATA POST INCREMENT**

With BIOS VERSION 1.5, you have the ability to use 256 bytes RAM of the AVRMEGA16 as a buffer to store temporary data. It's not eeprom, if the CPC booster+ resets, then the data are gone. With &FF28 you select the address, with &FF29 you read/write data to the buffer and with &FF2A, after reading/writing data, the address increases automatically by one.