The Byte Attic's

CERBERUS 2100™

© 2023 by
Bernardo Kastrup

Hardware Manual

Last edited: 7/26/23 4:20:11 PM

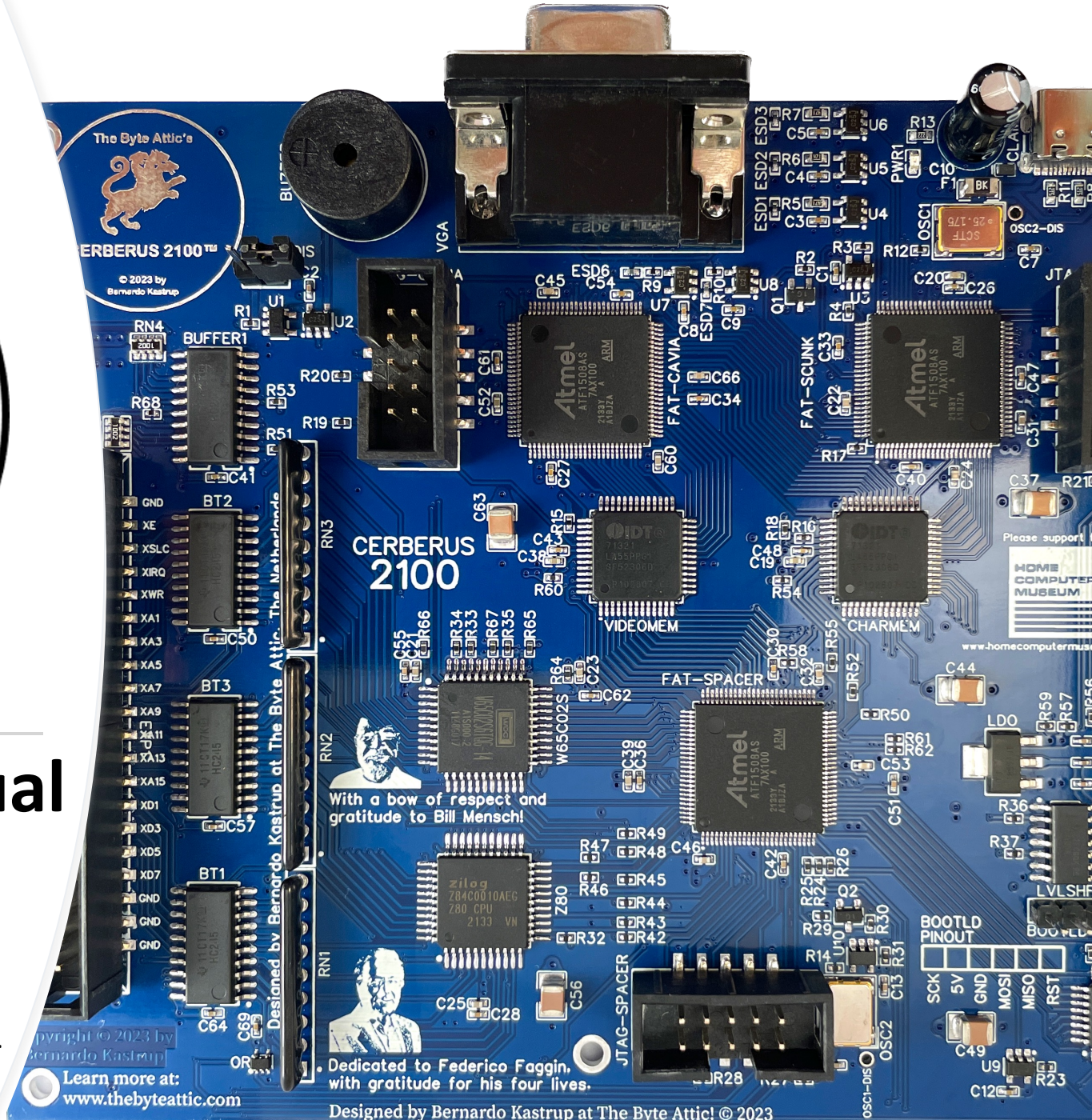


Table of contents

Introduction

- CERBERUS 2100™ is an innovative, *fully-functional computer*, not a toy
- It is an open-source *educational platform* for students of electronics and computer engineering
- It gives the user direct, convenient and unrestricted access to the hardware
- It allows for easy hardware changes and experimentation through in-system reprogramming of three CPLDs
- Its BIOS is an Arduino AVR sketch written in C, thus very easy to edit
- Its architecture is highly modular, allowing for compartmentalized, safe experimentation
- CERBERUS 2100™ aims to demystify computers by showing in detail how one is built, down to single gates and flip-flops
- Its architecture illustrates how a multi-processor system—with expansion possibility for even more processors—can be built
- Its design was done explicitly at the gate-level, with no high-level hardware synthesis tools
- CERBERUS 2100's architecture is clean and very easy to understand
- Two BASIC interpreters: one for the Z80 and the other for the W65C02S CPU

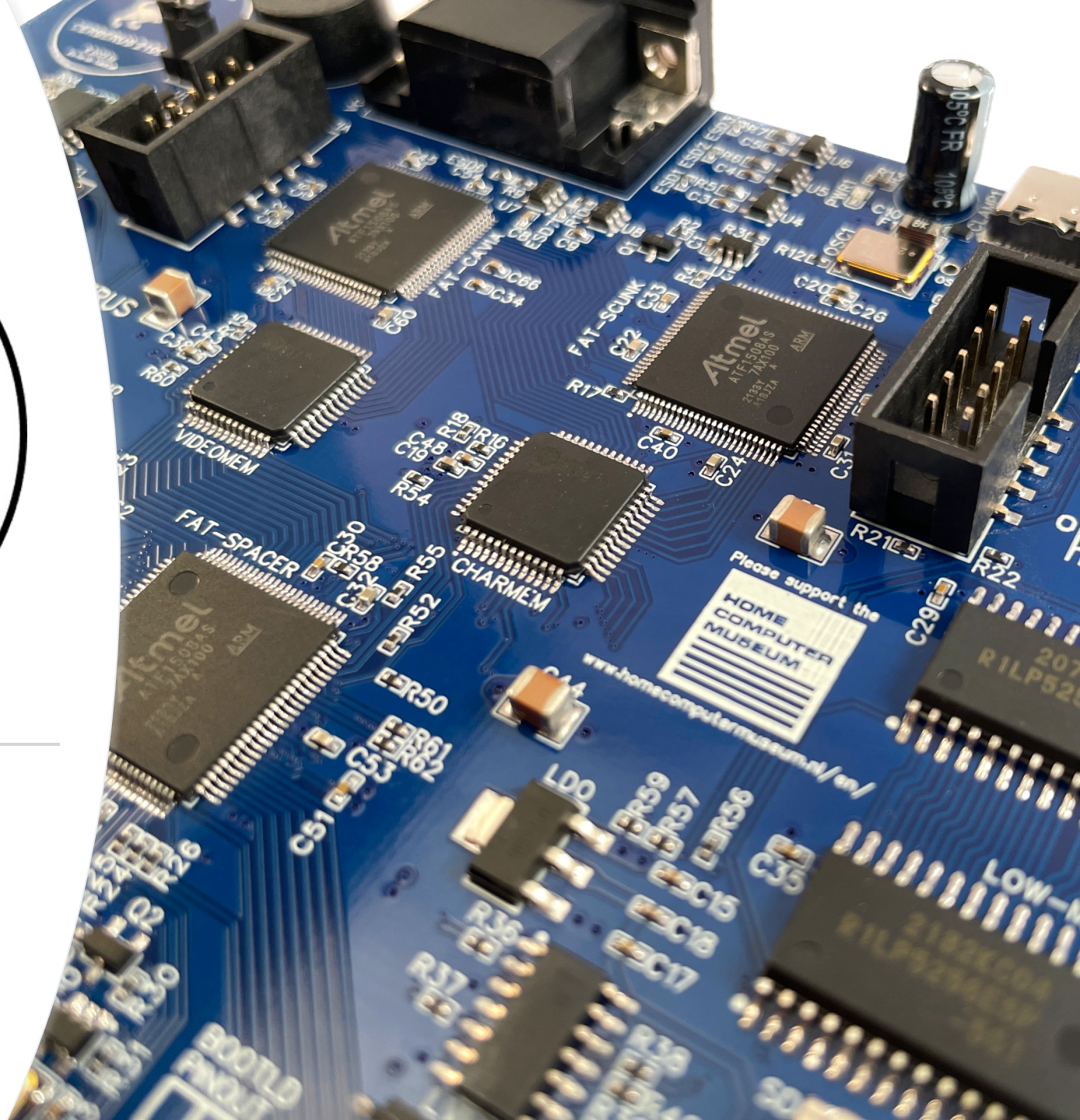
Technical specifications

- Complete 8-bit multi-processor microcomputer
- Built-in expansion slot with simple, generic I/O protocol
- 3 processors: Z80 and W65C02S CPUs, plus AVR I/O controller
- The CPUs run at 4 or 8 MHz (user-selectable), the AVR controller at 16 MHz
- Chipset with 3 custom ICs (CPLDs): FAT-SCUNK™, FAT-CAVIA™ and FAT-SPACER™
- Buzzer sound
- Standard PS/2-compatible USB keyboard
- Standard µSD card storage, with file system built into the BIOS
- 64 KB of user-addressable RAM
- No ROM: the BIOS is stored in the AVR controller's internal Flash and uses up no address space
- Standard VGA video, character-based, 320x240 pixels (40x30 individually addressable characters)
- On-the-fly user-redefinable character bitmaps for tile graphics

The Byte Attic's

CERBERUS 2100™

Technical overview



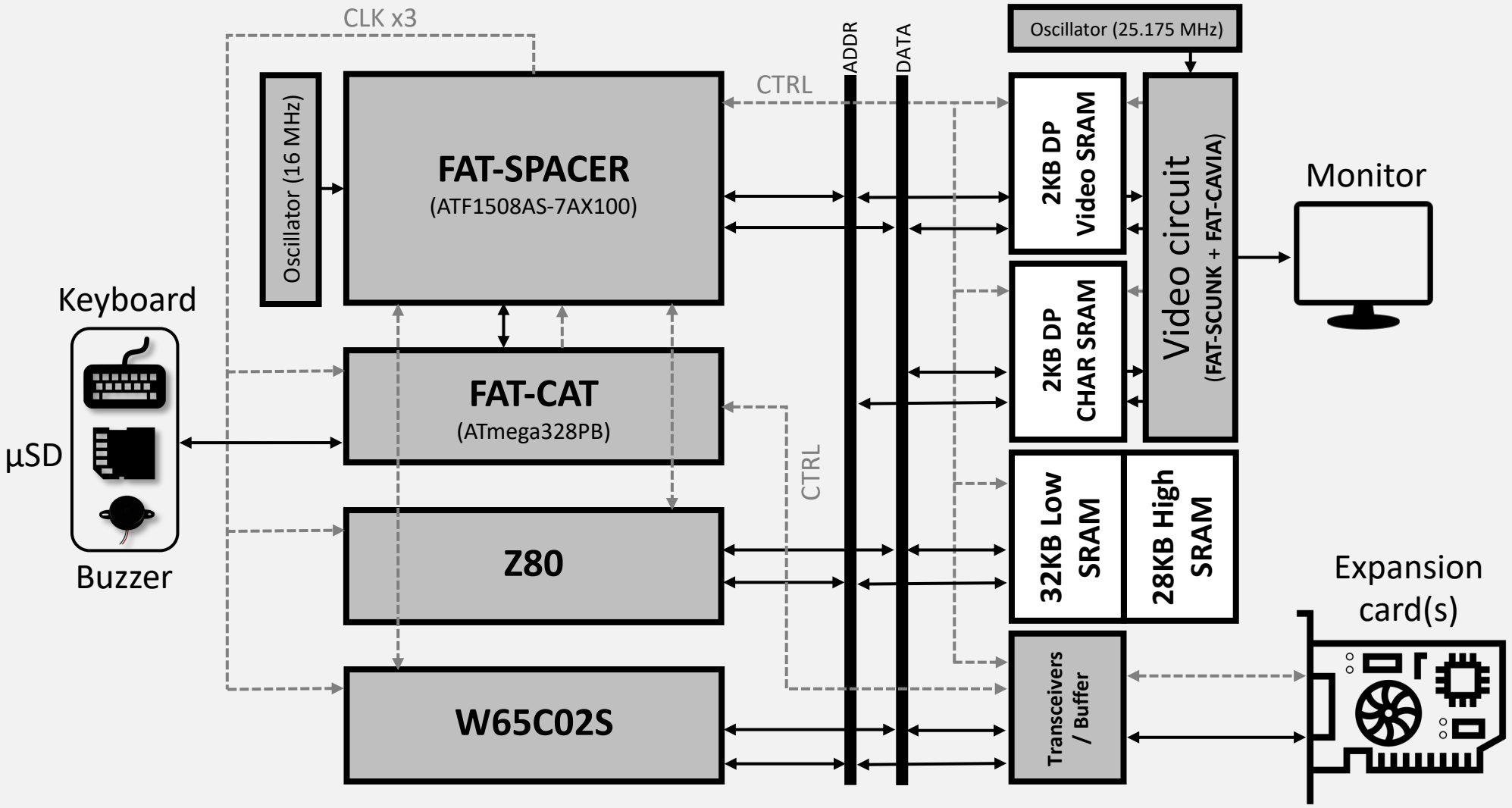
System overview

- CERBERUS 2100™ can be divided into three sub-systems:
  - The *video circuit*, driven by a 25.175 MHz oscillator
  - The *computer proper*, driven by a 16 MHz oscillator
  - The *expansion circuit*
- The three sub-systems are entirely asynchronous, modular, and communicate with each other via *memory-mapped I/O*
- The *expansion circuit* communicates with the *computer proper* via the two single-ported system memories, totaling 60 KB of addressable space
- The *computer proper* communicates with the *video circuit* via two dual-ported memories:
  - A 2KB *video memory*, storing a character identifier for each screen position
  - A 2KB *character memory*, which holds the character definitions or bitmaps
- Two custom ICs in the *video circuit:*
  - **FAT-SCUNK** (‘Scan CoUNter and clocK’), which controls all VGA timing
  - **FAT-CAVIA** (‘ChAracter Video Adapter’), which continuously scans the video and character memories to generate the screen

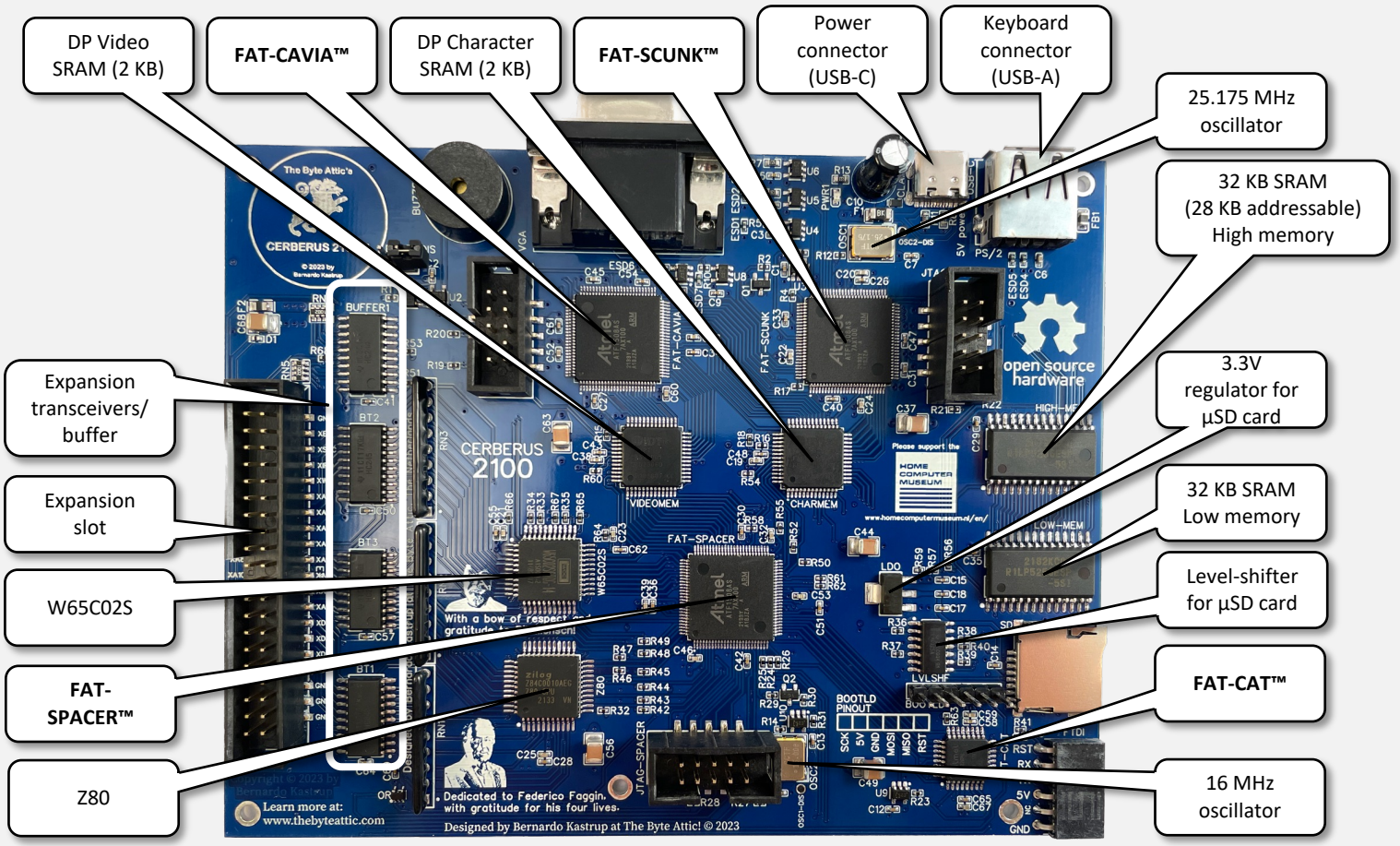
System overview (cont.)

- The *computer proper* has four main ICs:
  - **FAT-CAT** ('Custom ATmega328pb'), the I/O processor and system master
  - **FAT-SPACER** ('Serial to PArallel ControllER'), which manages all control signals, clocks, and translates FAT-CAT's serial data into parallel words & vice-versa
  - **Z80**, one of the two CPUs, responsible for running user code and a BASIC interpreter
  - **W65C02S**, the other CPU, also with its own BASIC interpreter
- As the system master, FAT-CAT runs the BIOS code (Basic Input/Output System) and delegates applications to the CPUs
- Two 32KB SRAMs (60KB user-addressable) serve as system memory
- No ROM: BIOS code is stored in FAT-CAT's internal Flash memory (32 KB) and *doesn't use system address space*
- The *expansion circuit* consists of:
  - Three tri-state bus transceivers to boost and isolate data & address buses
  - One buffer to boost and isolate control lines
  - A discrete logic gate for control signal processing
- Signals to/from the expansion slot are processed by FAT-CAT and FAT-SPACER

System architecture



Board overview



FAT-CAT™ overview

- An ATmega328PB microcontroller configured to use a strong external oscillator (16 MHz)
- FAT-CAT is CERBERUS 2100's *system master*: it runs the BIOS code from its onboard Flash memory and controls the CPUs and expansion
- The BIOS code is written in C and compiled under the Arduino IDE
- Except for video, FAT-CAT performs all I/O functions: file system operations, keyboard & expansion control, and sound output
- FAT-CAT determines the CPU clock frequency (4 or 8 MHz)
- Because of its serial nature, FAT-CAT is slow compared to the CPUs, but excels in flexibility and is therefore suitable for I/O operations & global system control
- FAT-CAT is capable of DMA (Direct Memory Access) through FAT-SPACER
  - Although FAT-CAT is a serial controller, through FAT-SPACER's internal shift registers it can access both data and address buses
- The user can only access the CPUs through FAT-CAT
  - It resets, selects, starts, halts, interrupts, passes on keyboard inputs & delegates applications to the CPUs via FAT-SPACER

FAT-SPACER™ overview

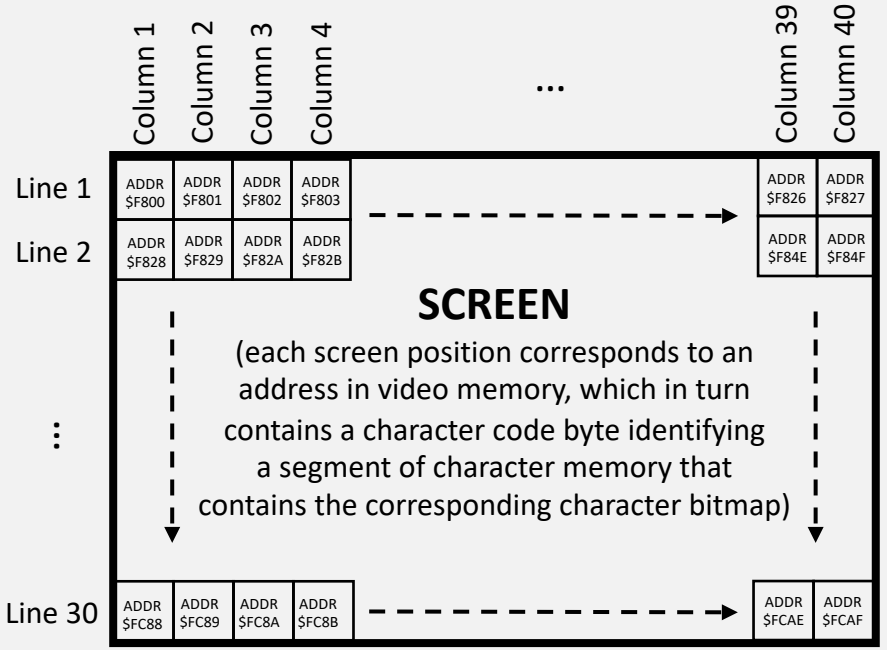
- FAT-SPACER is the glue that binds together the components of the *computer proper*
- Its design is the core of CERBERUS 2100's architecture, enabling the innovative multi-processor approach
- It replaces a standard control bus with a fully-connected control network that orchestrates the activity of the three processors and four memories
- All control inputs and outputs of all ICs in the *computer proper* are processed by FAT-SPACER's internal logic

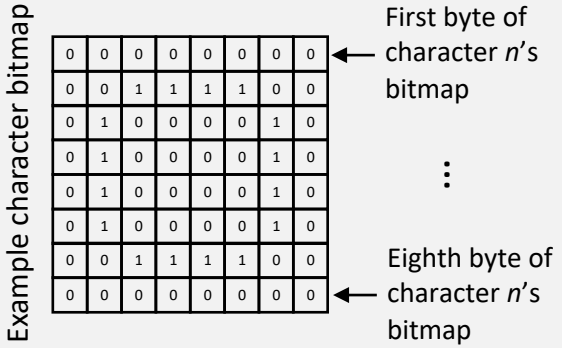
- FAT-SPACER's functions:
  - *Memory Selection*: enabling the appropriate memory IC depending on the contents of the address bus
  - *Expansion trigger:* generating the XSLC signal to trigger the expansion, depending on the contents of the address bus
  - *Read/Write control*: generating the appropriate write- and output-enable signals for the four memory ICs
  - *Clock Management*: generating and managing the clocks for FAT-CAT, the two CPUs and the expansion slot
  - *CPU Control*: generating the signals to reset, start, halt, tristate and interrupt the CPUs, based on input from FAT-CAT
  - *Serial/Parallel Translation*: translating FAT-CAT's serial data and addresses into parallel bus accesses, and vice-versa

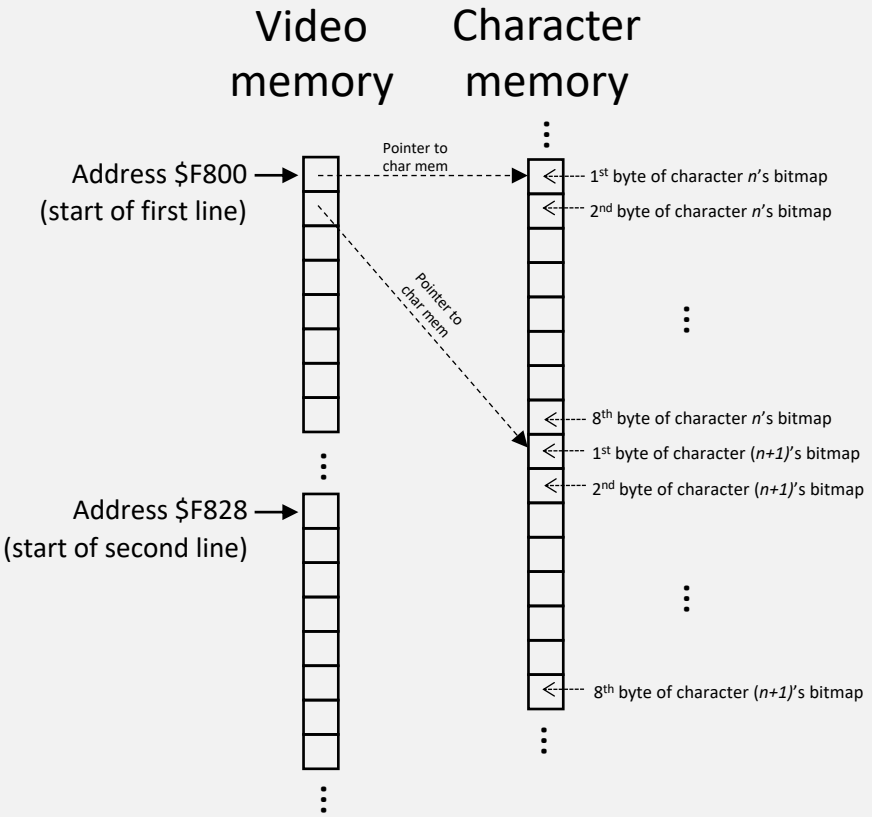
CERBERUS 2100™'s VGA mode

- CERBERUS 2100™ uses a standard VGA mode with 640x480 *screen pixels* and a 25.175 MHz pixel clock
- But each *CERBERUS pixel* is a 2x2 matrix of *screen pixels*
- CERBERUS 2100's resolution is thus 320x240 pixels, or 40x30 characters of 8x8 pixels each
- This translation is achieved by sampling each CERBERUS pixel *four times: t*wice horizontally and twice vertically
- Each of the 40x30 addresses in video memory holds a byte that identifies a character from the character set
- FAT-CAVIA reads out that byte, which then becomes the 8 most-significant bits of an 11-bit character memory address
- FAT-CAVIA reads out the 8 bytes of the corresponding character bitmap in character memory by progressively incrementing the 3 least-significant bits of the address from 0 to 7

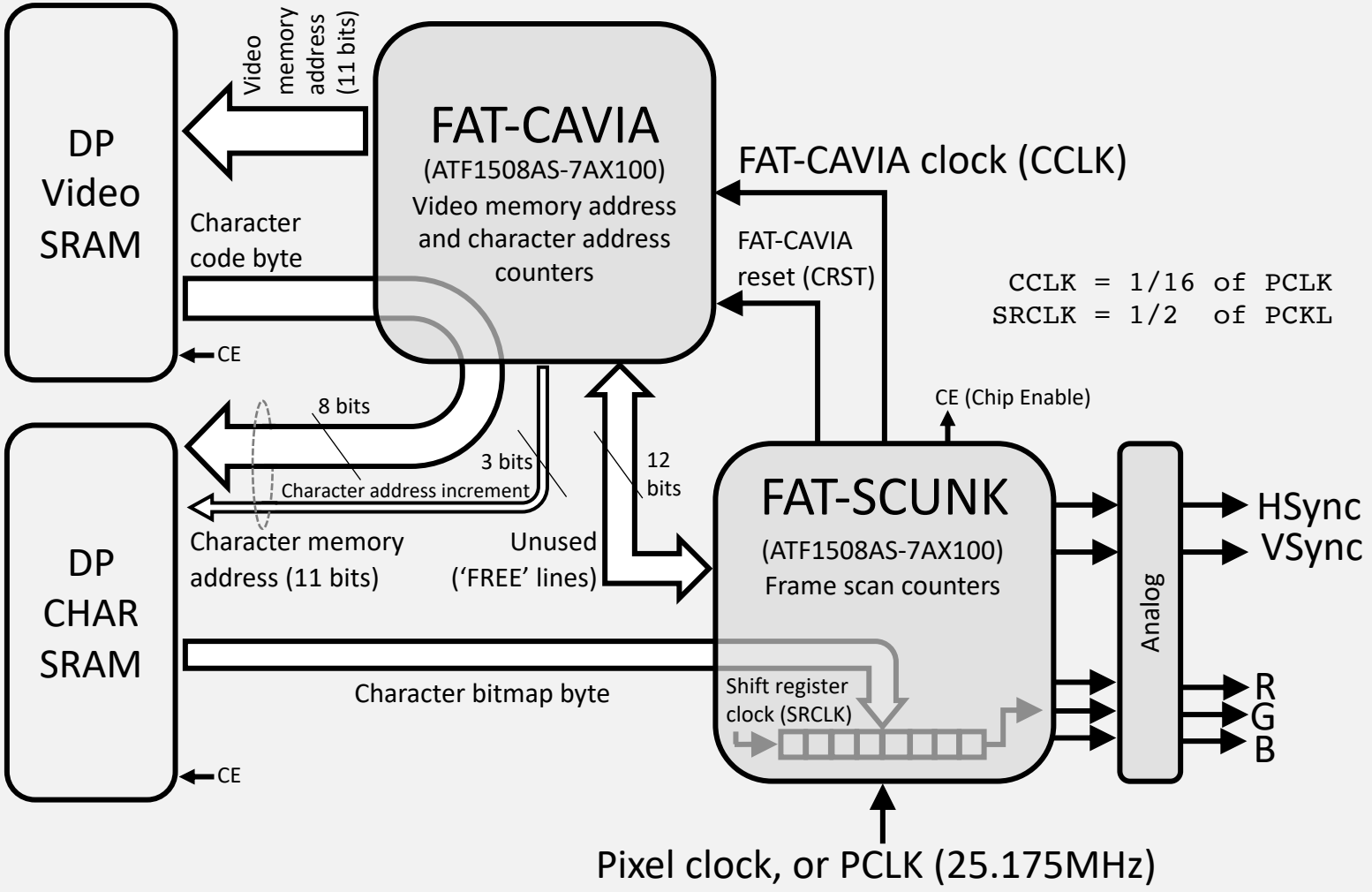
Video and character memory organization







Video circuit architecture



FAT-CAVIA™ overview

- FAT-CAVIA continuously scans all addresses of video memory, as well as the corresponding addresses in character memory
- Each 40-character line in video memory is scanned *16 times* per frame
    - 8 bytes per character times 2 passes per byte, so to double-sample it vertically
- Each read from video memory produces a character code byte that constitutes the 8 most-significant bits of the corresponding 11-bit character memory address
- FAT-CAVIA uses a counter to produce the 3 least-significant bits of the 11-bit address, so to scan all 8 bytes of the corresponding character bitmap in character memory
- Since each character memory read leads to 8 pixels (i.e. a byte from the character bitmap), which are then double-sampled horizontally, FAT-CAVIA's clock (CCLK) is 1/16 of the pixel clock (PCLK)

FAT-SCUNK™ overview

- Each video frame consists of *480 pixel lines* and is followed by a *vertical blanking interval* equivalent in timing to an additional 45 lines
- Each pixel line consists of *640 screen pixels* followed by a *horizontal blanking interval* equivalent in timing to an additional 160 screen pixels
- Both the vertical and horizontal blanking intervals encompass *synchronization pulses*
- FAT-SCUNK is responsible for counting all the relevant intervals and producing the synchronization pulses
- It also generates the FAT-CAVIA clock (CCLK) from the pixel clock (PCLK) using an internal clock divider
- It features an internal 8-bit shift register, wherein each byte read out from character memory is temporarily stored
- It then shifts each bit of that byte out to the RGB lines of the VGA connector, according to a shift register clock (SRCLK)
- SRCLK is generated internally by FAT-SCUNK, through a clock divider, and is ½ of PCLK so to horizontally double-sample each bit

FAT-SCUNK: action timing

States of internal counter

S_3	S_2	S_1	S_0	Action taken by FAT-SCUNK
0	0	0	0	
0	0	0	1	Load current character byte into shift register
0	0	1	0	CAVIA clock tic to go to next character byte
0	0	1	1	Shift right
0	1	0	0	
0	1	0	1	Shift right
0	1	1	0	
0	1	1	1	Shift right
1	0	0	0	
1	0	0	1	Shift right
1	0	1	0	
1	0	1	1	Shift right
1	1	0	0	
1	1	0	1	Shift right
1	1	1	0	
1	1	1	1	Shift right

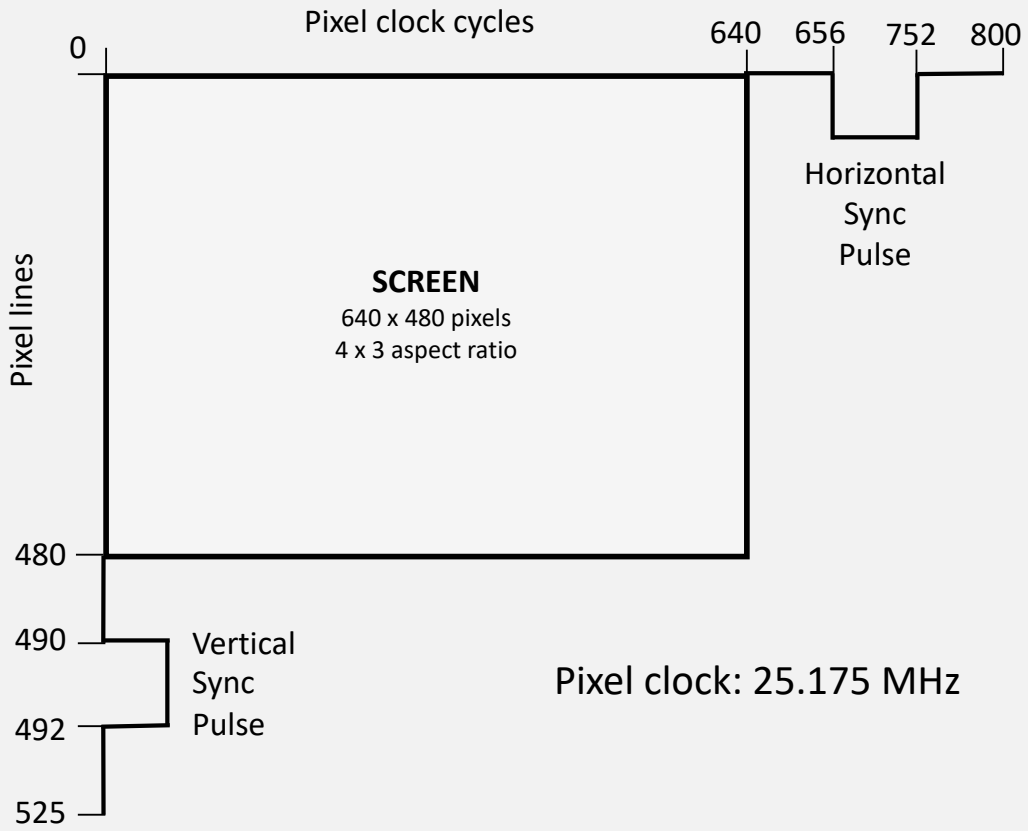
Corresponding hardware description
(see `FAT-SCUNK.PLD` file)

```
CCLK = !S3 & !S2 & S1 & !S0;

ShiftRegister.d =  (!S3 & !S2 & !S1) & CharByteMirrored #
                   !(!S3 & !S2 & !S1) & ShiftOneRight;

ShiftRegister.ck = S0;
```

Standard VGA signal



Colors

- From the programmer's perspective, CERBERUS 2100™ only has two colors
  - Character bitmaps are binary
  - Software cannot select colors
- In the default configuration, FAT-SCUNK defines these colors as blue (background) and yellow (foreground)
  - This is done in the following lines of the file FATSCUNK.PLD, available in the distribution:

```
RED   = OUTBIT;
GREEN = OUTBIT;
BLUE  = NB & !OUTBIT;
```

- In the code shown, '`OUTBIT`' is the output of the shift register containing a byte from the current character's bitmap, '`NB`' ('Non-Blanking') is active outside the blanking intervals, '`&`' is a logical AND, and '`!`' is a logical inversion
- The code thus translates into: "red and green together (i.e., yellow) are active when the output bit of the shift register is 1, while blue is active when such bit is 0 while outside any blanking interval"

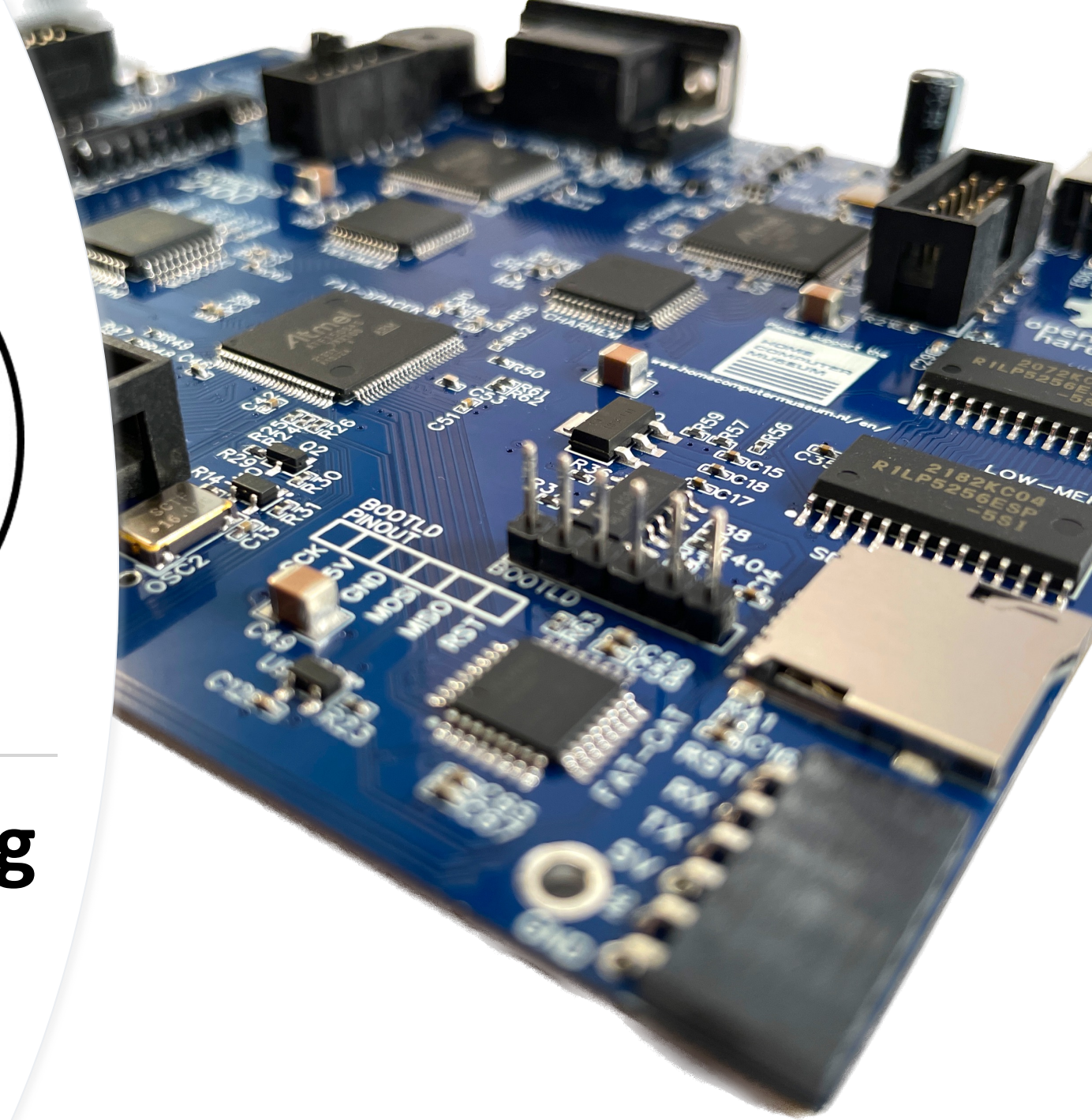
Color maps and colored characters

- **Color maps** are possible: FAT-SCUNK can change what the two colors are for each *position on the screen*
  - There are 9 unused I/O lines connecting FAT-CAVIA to FAT-SCUNK (FREE4 to FREE12), so the former can pass on the *address in video memory* currently being scanned
  - This address can be used in the logic equations for signals RED, GREEN and BLUE
  - A given screen position will then have the two colors selected, regardless of what character occupies it
  - This will, of course, require reprogramming FAT-SCUNK, but is a fun way to experiment with programming CPLDs, as the results are immediately visible on the screen
- **Colored characters** are also possible: FAT-SCUNK can change what the two colors are for each *character code*
  - There are 9 unused I/O lines connecting FAT-CAVIA to FAT-SCUNK (FREE4 to FREE12), so the former can pass on the *character code* currently being scanned
  - Like before, this address can be used in the logic equations for signals RED, GREEN and BLUE
  - A given character will then have the two colors selected, regardless of where in the screen it is located
  - Again, this will require reprogramming FAT-SCUNK, but is also a fun way to experiment with programming CPLDs, as the results are immediately visible on the screen

The Byte Attic's

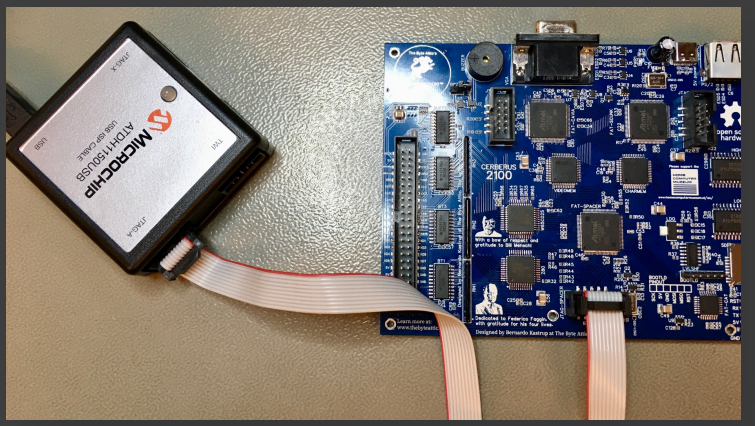
CERBERUS 2100™

Programming instructions



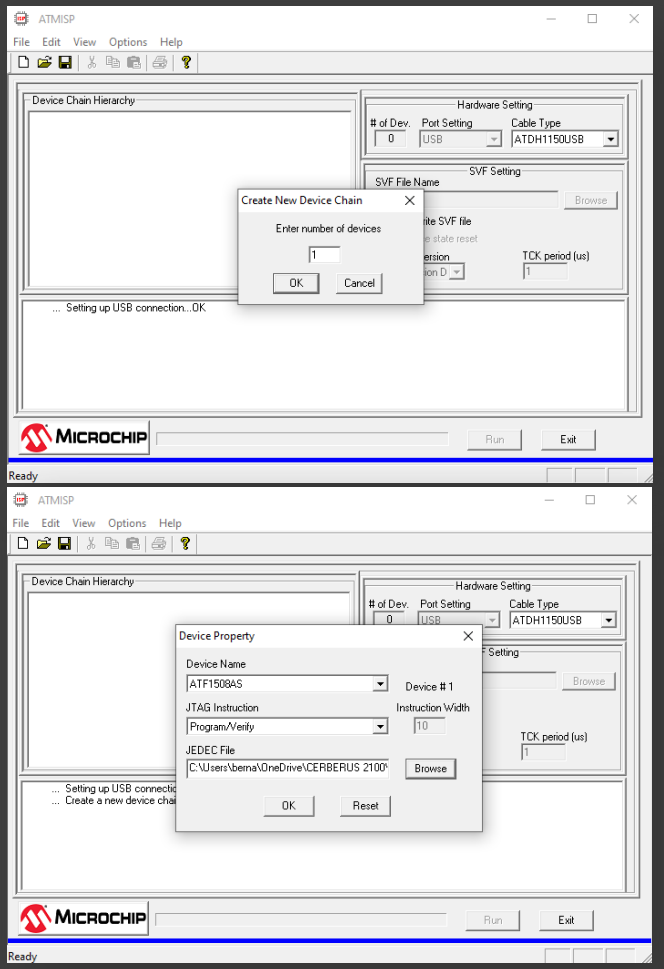
(Re)programming the CPLDs: preparation

1. If you bought a ready-to-use CEBERUS 2100™ and are not modifying the CPLD circuits, you may skip this and the next page
2. Otherwise, you will need the following tools:
   - Microchip **ATDH1150USB** JTAG USB ISP cable
   - Microchip **ATMISP v7.3** or later, freely-downloadable in-system programing software: https://www.microchip.com/en-us/products/fpgas-and-plds/spld-cplds/pld-design-resources
3. Connect one end of the USB cable that accompanies the ATDH1150USB to the 'USB' port of the JTAG box, and the other end to your Windows PC
4. Connect one end of the 10-wire flat cable that accompanies the ATDH1150USB to the 'JTAG A' port of the JTAG box (see picture)



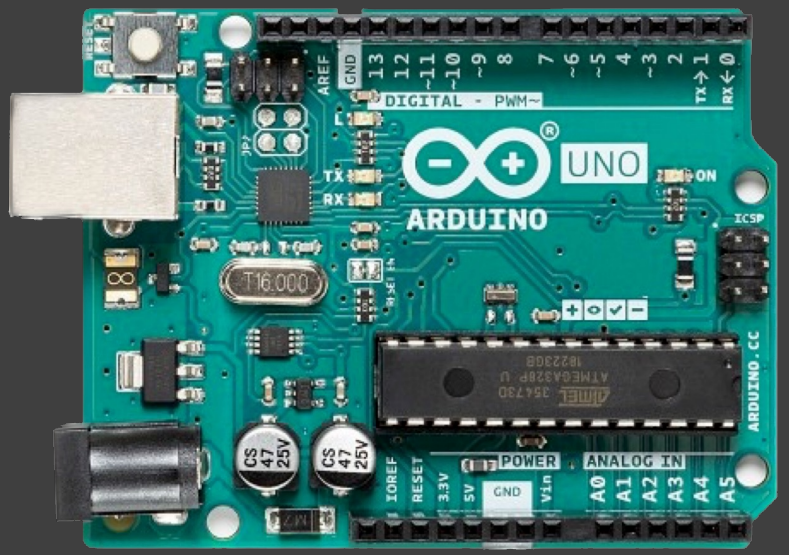
(Re)programming the CPLDs

5. Connect the free end of the 10-wire flat cable to the appropriate JTAG header on the CERBERUS 2100™ board (each of FAT-SCUNK, FAT-CAVIA and FAT-SPACER has a marked 10-pin JTAG header close to it)
6. Connect CERBERUS 2100™ to a USB power source via the USB-C connector and turn it on
7. In your Windows PC, open ATMISP and click: `File → New`
8. In the pop-up window, enter 1 device and click `OK`
9. In the new pop-up window, under `Device Name` choose: `ATF1508AS`
10. Under `JTAG Instruction` choose: `Program/Verify`
11. Next to `JEDEC File` click `Browse` and choose the appropriate .jed file from CERBERUS 2100's distribution (e.g. choose FAT-SPACER.jed to program FAT-SPACER), click `OK`
12. Click `Run` and wait for completion
13. Turn CERBERUS 2100™ off
14. Repeat the above for FAT-SCUNK, FAT-CAVIA and FAT-SPACER, repositioning the 10-wire flat cable accordingly each time



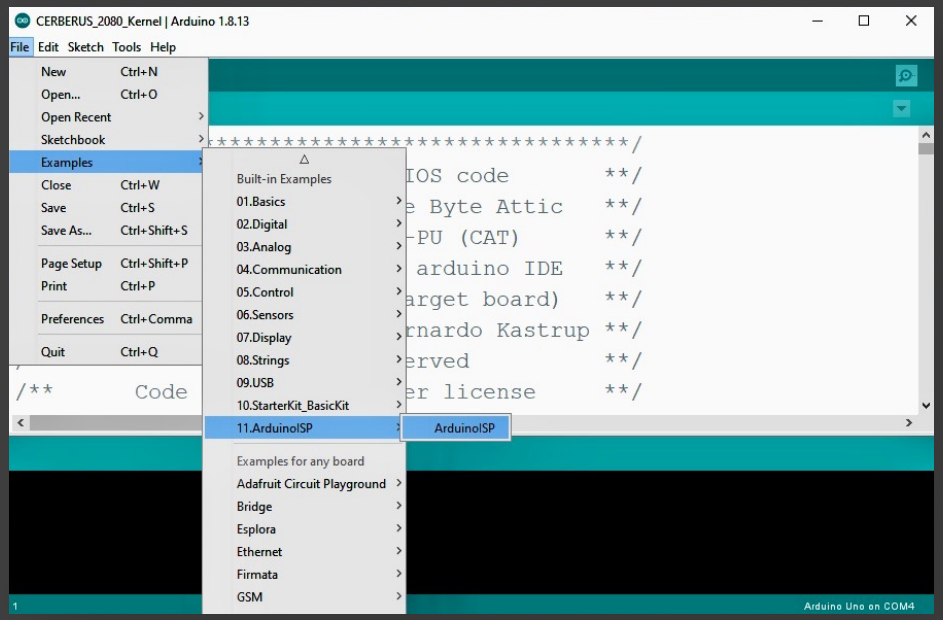
Programming FAT-CAT: preparation

1. If you bought a pre-programmed CERBERUS 2100™, it already has a bootloader burned in it, so you may skip ahead to step 18
2. Otherwise, you will need:
   - An Arduino UNO board (with USB cable)
   - An Arduino IDE installation on your PC: https://www.arduino.cc/en/software
   - 5 male-to-female jumper wires
   - A 10µF capacitor rated for 6.3V or higher
3. Make sure FAT-SPACER is already programmed, for without it you won't be able to burn the bootloader, as FAT-CAT won't be getting a clock



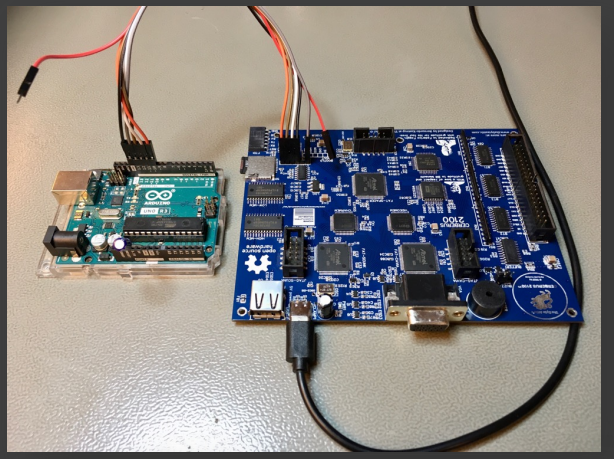
Burning the bootloader

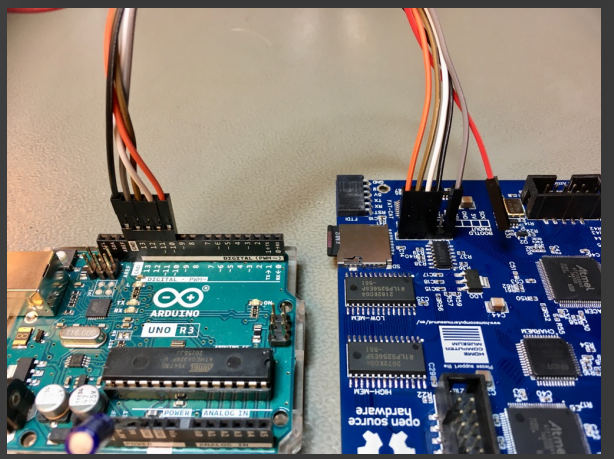
4. Connect the Arduino UNO to your computer (via USB) and launch the Arduino IDE
5. In the Arduino IDE, load: `File → Examples → 11.ArduinoISP → ArduinoISP`
6. Compile and upload the `ArduinoISP` sketch to the Arduino UNO
7. When done, disconnect the Arduino UNO from your PC



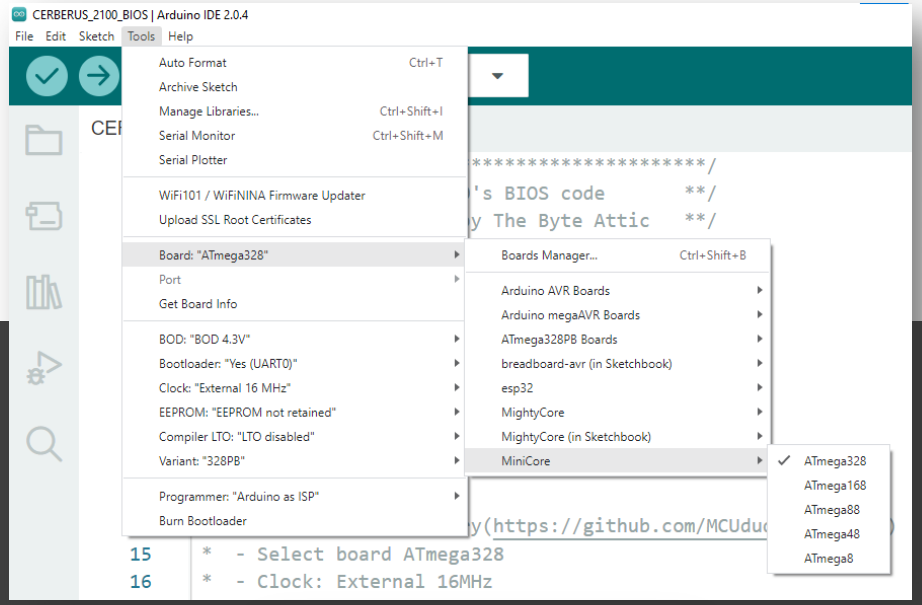
Burning the bootloader (cont.)

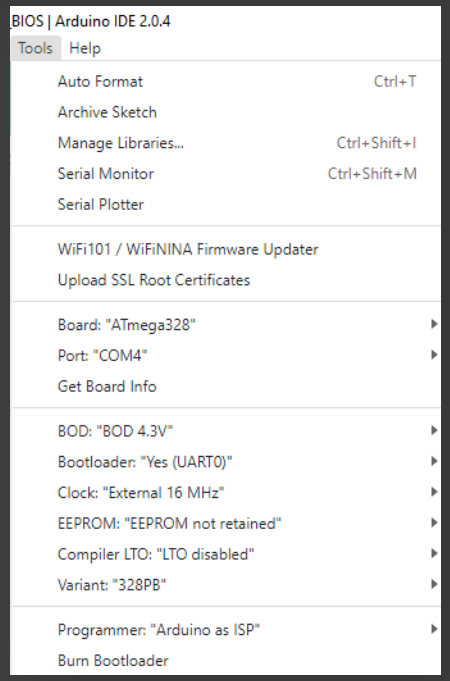
8. Set up the cables as shown in the pictures
   - Connect the 10µF capacitor between GND and RESET of the Arduino UNO (watch out for the polarity!)
   - Connect pins 13 to 10 of the Arduino Uno respectively to pins SCK, MISO, MOSI, and RST of the ‘BOOTLD’ header on the CERBERUS 2100™ board
   - Connect GND on the Arduino Uno to the GND pin of the BOOTLD header
   - You do *not* need to connect the 5V line
9. Install MCUdude’s MiniCore library in the Arduino IDE, following the instructions provided on the library’s repository at: https://github.com/MCUdude/MiniCore
10. Reconnect the UNO to your PC





Burning the bootloader (cont.)

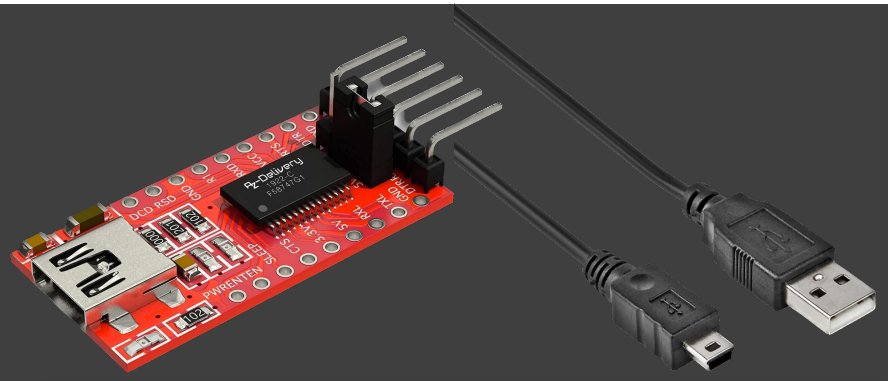


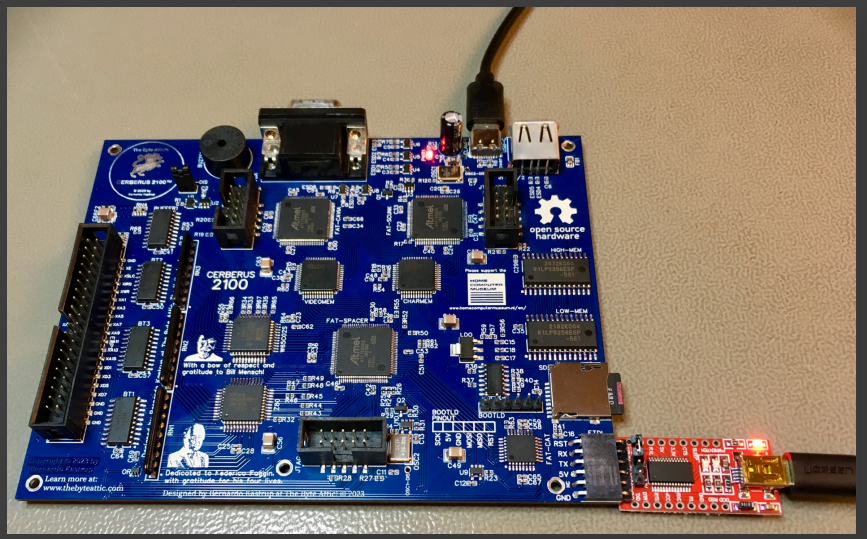


11. From the Arduino IDE, select:
    `Tools → Board → MiniCore → ATmega328`
12. Set the options as per the picture to the right
13. Still from the Arduino IDE, change the COM port to the one active in your case:
    `Tools → Port →` (active port)
14. Now select:
    `Tools → Programmer → Arduino as ISP`
15. Connect CERBERUS 2100™ to a USB power source via the USB-C connector and turn it on
16. And finally:
    `Tools → Burn Bootloader`
17. After completion, turn CERBERUS 2100™ off and disconnect the Arduino UNO from your PC

(Re)programming the BIOS

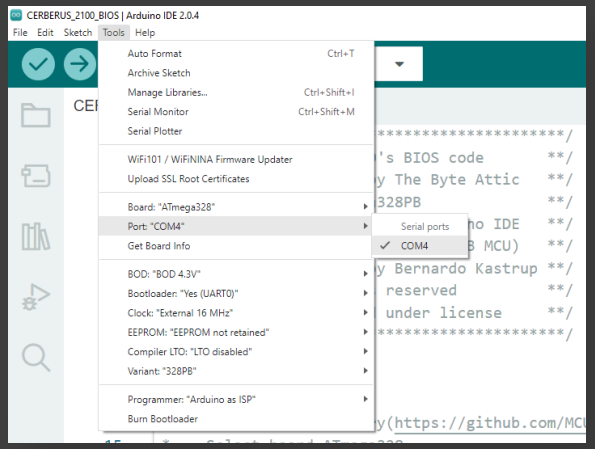
18. You will now need:
    - An AZDelivery FTDI Adapter FT232RL USB to TTL Serial (or equivalent)
    - A standard USB-A to mini-USB cable
19. Remove the voltage jumper from the FTDI adapter (no voltage selected)
20. Connect the FTDI adapter to your PC via the USB cable
21. Connect the FTDI adapter to the FTDI port of the CERBERUS 2100™ board, facing up
    (see picture to the right)
22. In the Arduino IDE, open the `CERBERUS_2100_BIOS.ino` sketch in the directory `CAT/CERBERUS_2100_BIOS/` of the CERBERUS 2100™ distribution
23. Make sure the directory `src/` of the distribution is next to the sketch
24. Connect CERBERUS 2100™ to a USB power source via the USB-C connector and turn it on





(Re)programming the BIOS (cont.)

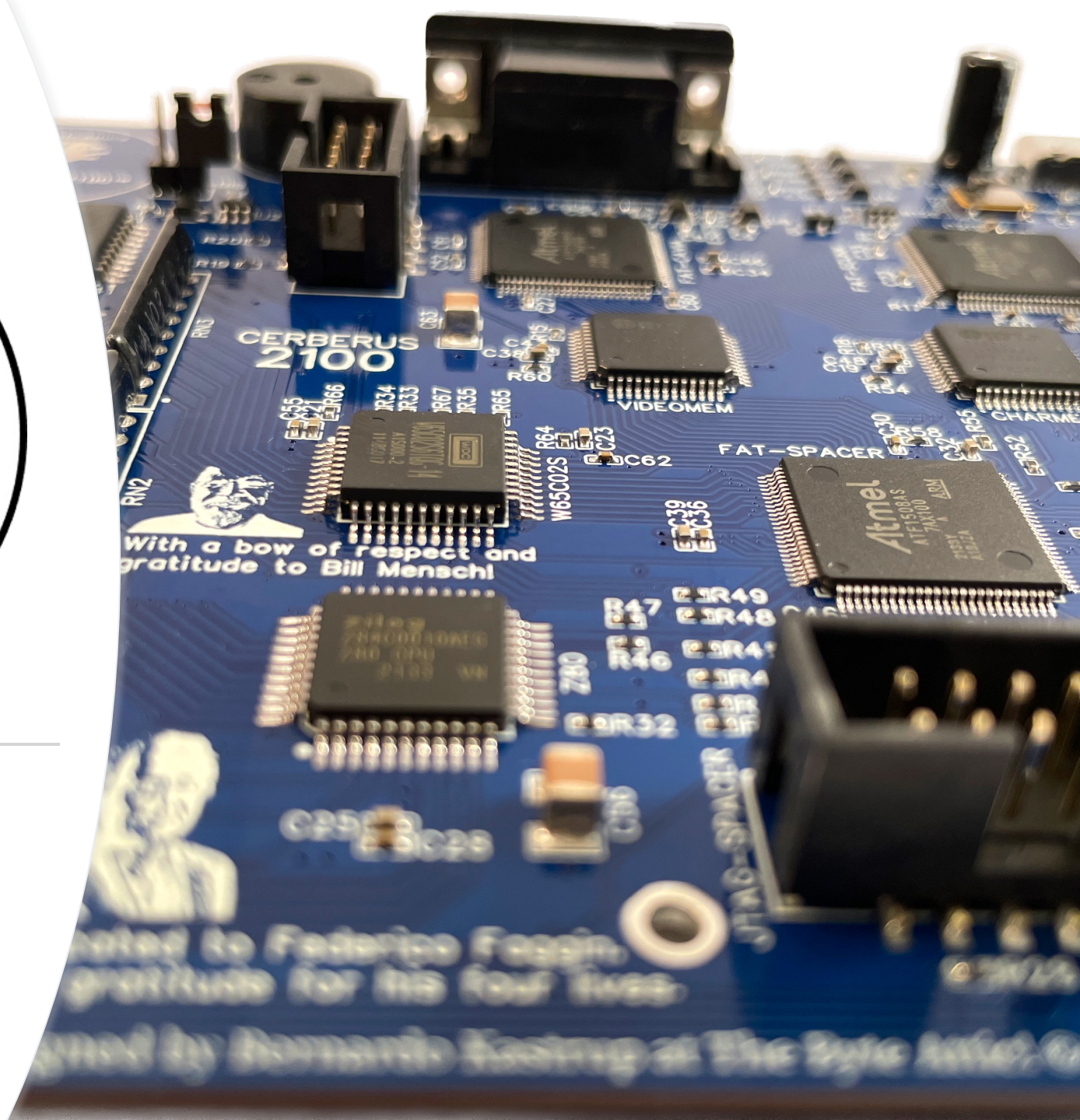
25. Select the now-active COM port:
    `Tools` → `Port` → (active port)
26. Compile and upload the sketch
    (you may now hear CERBERUS 2100™ beep, which is normal; if it annoys you, disable the buzzer by placing a jumper on the BUZZ-DIS header)
27. Turn CERBERUS 2100™ off
28. Disconnect the FTDI adapter from the CERBERUS 2100™ board
29. Copy the files in the `Ceberus uSD card files/` directory of the distribution to a FAT32-formatted, class-10 (or higher) µSD card
30. Insert the µSD card into CERBERUS's µSD card adapter
31. Connect CERBERUS 2100™ to a VGA monitor, PS/2-compatible keyboard and a USB power source
32. Turn CERBERUS 2100™ on
33. You're done! CERBERUS 2100™ should now boot normally and display the start-up screen shown in the picture to the right



The Byte Attic's

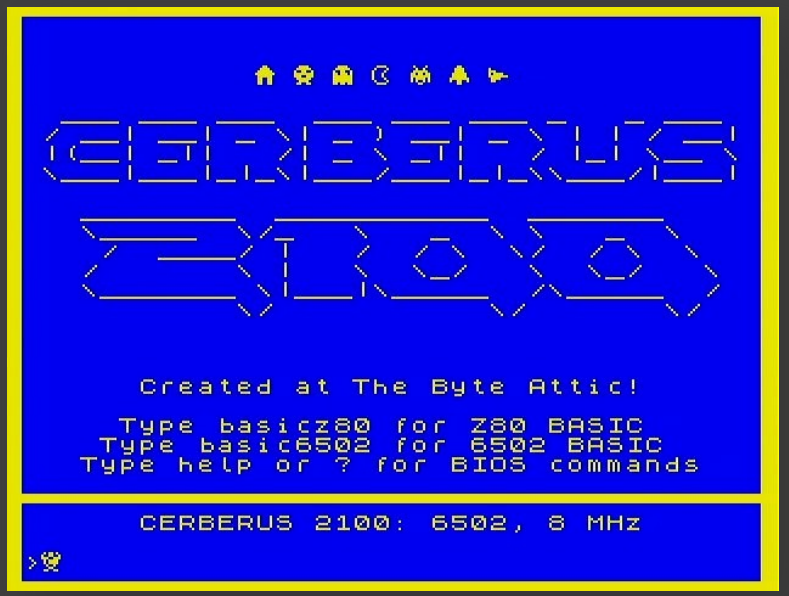
CERBERUS 2100™

User's manual



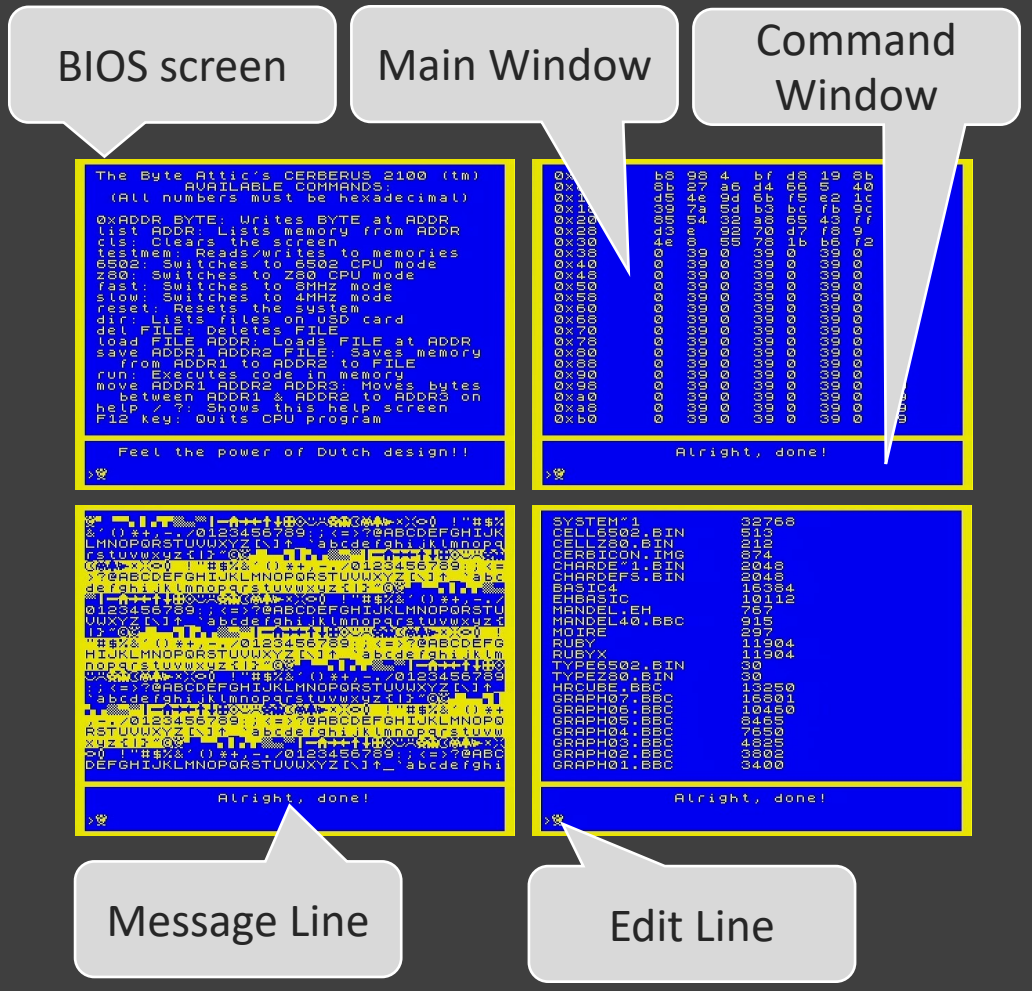
Powering CERBERUS 2100™ on

1. To boot properly, CERBERUS 2100™ needs to have a µSD card inserted in it, loaded with the files in the `Ceberus uSD card files/` directory of the distribution
2. If CERBERUS 2100™ finds no µSD card, it will produce a repeating beep to indicate an error, and show garble on the screen
3. If a µSD card is found, but lacks the `chardefs.bin` file, CERBERUS 2100™ will beep once and display garble
4. The correct startup screen is shown in the picture



The BIOS screen

- The user interacts with CERBERUS 2100™ through the *BIOS Screen*, which is displayed upon startup
- The BIOS Screen has two segments: the *Main Window* above and the *Command Window* below
- The Command Window is further divided into two segments: the *Message Line* above and the *Edit Line* below
- CERBERUS 2100™ displays system messages to the user in the Message Line
- The user gives CERBERUS 2100™ commands by typing them out in the Edit Line
- Type ‘help’ or ‘?’ to display a list of the commands available, which will be discussed in more detail in the next pages



Commands: basics

- Typing the **`up-arrow`** fills the Edit Line with the last command typed
- Typing the **`down-arrow`** clears the Edit Line
- **`run`** executes the code currently in the code area (starting at address `$0205`) on the selected CPU
  - If the code in memory is meant for one of the CPUs, but you try to execute it on the other, the result will be unpredictable
- If the CPU is running code, press **`F12`** to return to the BIOS Screen
- **`cls`** clears the Main Window
- **`reset`** resets CAT and the two CPUs, redraws the startup screen, but does *not* otherwise erase the contents of memory
- **`6502`** selects the W65C02S CPU
- **`z80`** selects the Z80 CPU
- **`fast`** sets the CPU clock at 8 MHz
- **`slow`** sets the CPU clock at 4 MHz
- **`help`** or **`?`** displays a summary of the commands available

Commands: file manipulation

- All numbers typed into the Edit Line are assumed to be hexadecimal
  - Instead of typing '`0xFF`' or '`$FF`', type simply '`FF`'
- File names are *not* case-sensitive, so '`CODE.bin`' and '`code.bin`' will be the same file
- **`load FILE ADDR`** loads the contents of a binary file named `FILE` from the µSD card into memory, starting from the address `ADDR` (in hex)
  - If `ADDR` is not provided, CERBERUS 2100™ will default to `$0205`, the start of the code area
  - The file name `FILE` needs to be typed out in full
  - Example: '`load cell6502.bin`'
- **`save ADDR1 ADDR2 FILE`** saves the contents of memory from `ADDR1` to `ADDR2`, inclusive, to a binary file named `FILE` in the µSD card
  - Example: '`save 205 2ff program.bin`'
- **`del FILE`** deletes the file named `FILE` from the µSD card
  - The name `FILE` needs to be typed out in full
  - Example: '`del typez80.bin`'
- **`dir`** lists the files on the µSD card
  - CERBERUS's file system, built into the BIOS, does *not* support sub-directories, so all files must be in the root of the µSD card

Commands: editing memory

- **`list ADDR`** displays the contents of memory from address `ADDR` onwards, until the Main Window is filled up
  - If `ADDR` is not provided, CERBERUS 2100™ will default to `$0000`
  - Example: '`list F800`'
- Again, all numbers typed into the Edit Line are assumed by CERBERUS 2100™ to be hexadecimal
  - Instead of typing '`0xFF`' or '`$FF`', type simply '`FF`'

- **`0xADDR BYTE(s)`** inserts the list of bytes `BYTE(s)` (wherein individual bytes are separated by spaces or commas) into memory, starting from address `ADDR`
  - The list of bytes can be as long as it fits in the Edit Line, or have a single byte
  - You can use this command to write directly into video and character memories, which is handy to alter and see the character definitions
  - Try out these cool examples (after '`cls`'):
    '`0xF82A 00 01 02 03 04 05 06 07`'
    '`0xF000 FF 00 FF 00 FF 00 FF 00`'

Commands: editing memory (cont.)

- **`move ADDR1 ADDR2 ADDR3`** copies the contents of memory in the segment between `ADDR1` and `ADDR2`, inclusive, to the segment starting at `ADDR3`
  - Example:
    '`move 0000 00FF FF00`'

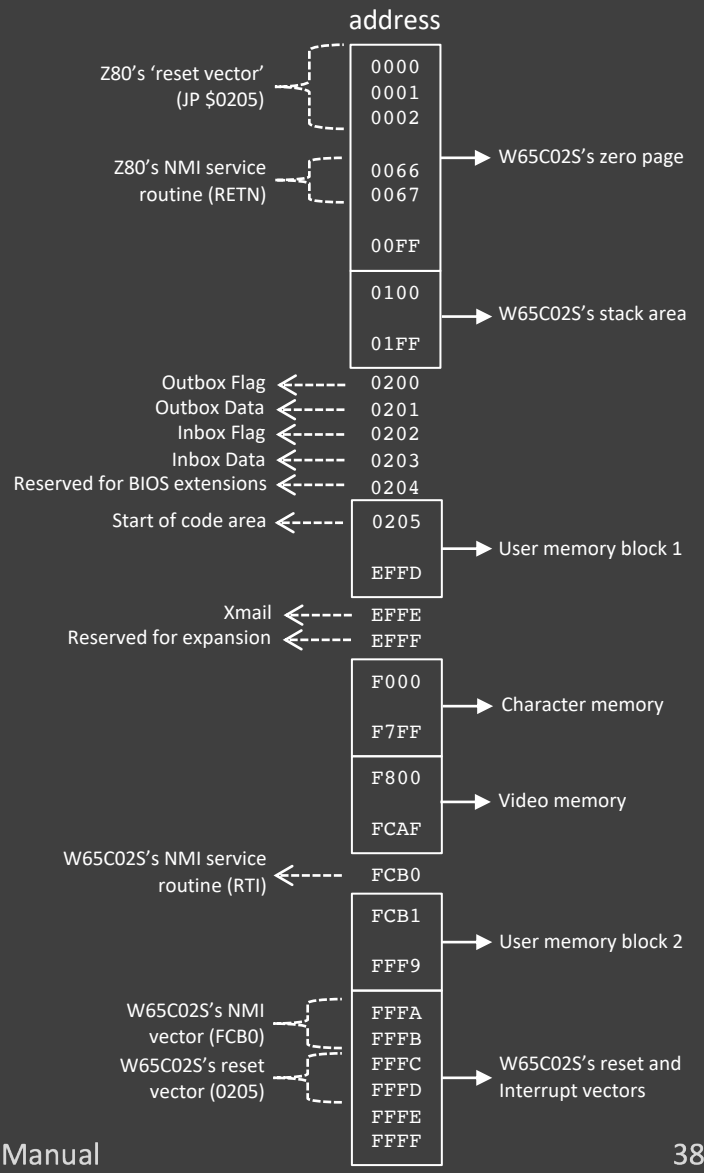
- **`testmem`** writes a sequence of numbers into low memory, then reads it from low memory and writes it to high memory, then reads it from high memory and writes it to video memory, where the sequence is interpreted as characters and displayed
  - This is a *non-exhaustive* test of the memory subsystem
  - It also displays the character set, as currently defined in character memory, on the screen

The character set

- CERBERUS 2100™ allows for 256 different character definitions, each identified by an extended ASCII value or character code, ranging from 0 to 255
- The picture illustrates the default character set, listed from 0 to 255 from the top left
  (it repeats itself after it ends)
- The character definitions are stored in addresses $F000 to $F7FF (inclusive)
- Each character is an 8x8 bitmap, and thus requires 8 bytes in its definition
- The bitmap of character 0 (the cursor character) is stored in addresses $F000 to $F007
- The bitmap of character 1 is stored in addresses $F008 to $F00F, and so on
- All character definitions can be modified by the user from the BIOS Screen or dynamically, from a program running on one of the CPUs
- Try this command, which changes the cursor:
  '`0xF000 FF 00 00 00 00 00 00 FF`'
- To reload the standard definitions again, type:
  '`reset`' or '`load chardefs.bin F000`'

The memory map

- The diagram shows CERBERUS 2100™'s memory map
- Address $0205 is the start of the code area
- Addresses $0200 and $0202 are *flags*: FAT-CAT and CPU set these flags to inform each other that a byte is being passed
- Addresses $0201 and $0204 are *data boxes*: they contain the actual byte that is being passed between FAT-CAT and CPU
- When the address *Xmail* ($EFFE) is written to, FAT-SPACER pulls XSLC (see Expansion Slot section ahead) low for a cycle, to alert the expansion card that there is data for it in memory
- Any byte poked into the video area will be interpreted as a character code and the corresponding character bitmap will be displayed on the screen
- Any byte poked into the character memory will be interpreted as an update to a character definition
- The largest contiguous area available for programs and data is the user memory block 1, which resides physically in the two 32KB SRAM ICs
- The user memory block 2 is also available, and resides physically in the DP Video SRAM (the screen does not use all the addresses in this chip)



Test programs

- CERBERUS 2100™'s distribution contains four test assembly programs:
  - `TYPE6502.bin`
  - `TYPEZ80.bin`
  - `CELL6502.bin`
  - `CELLZ80.bin`
- The corresponding source (`.asm`) files are also provided in the in the directory `CERBERUS Applications Source Code/` of the distribution
- The 'TYPE' application simulates a typewriter on the screen: it prints out, in sequence, the keys you press on the keyboard
  - It tests the interface between FAT-CAT (which reads the keyboard) and the active CPU (which receives the character code read via memory-mapped I/O) during application execution
- The CELL application is a Wolfram Rule-30 linear cellular automaton that fills the screen with fractal, vertical-scrolling patterns
  - It tests the speed of CERBERUS 2100's software-based scrolling
- The `CELL6502.bin` code is slightly more sophisticated then the `CELLZ80.bin` code
  - The W65C02S version also updates the character definitions in the character memory on-the-fly, during execution
- The BIOS will not prevent you from trying to run W65C02S code on the Z80, or Z80 code on the W65C02S
  - If you accidently do so, the result is unpredictable, but won't damage anything

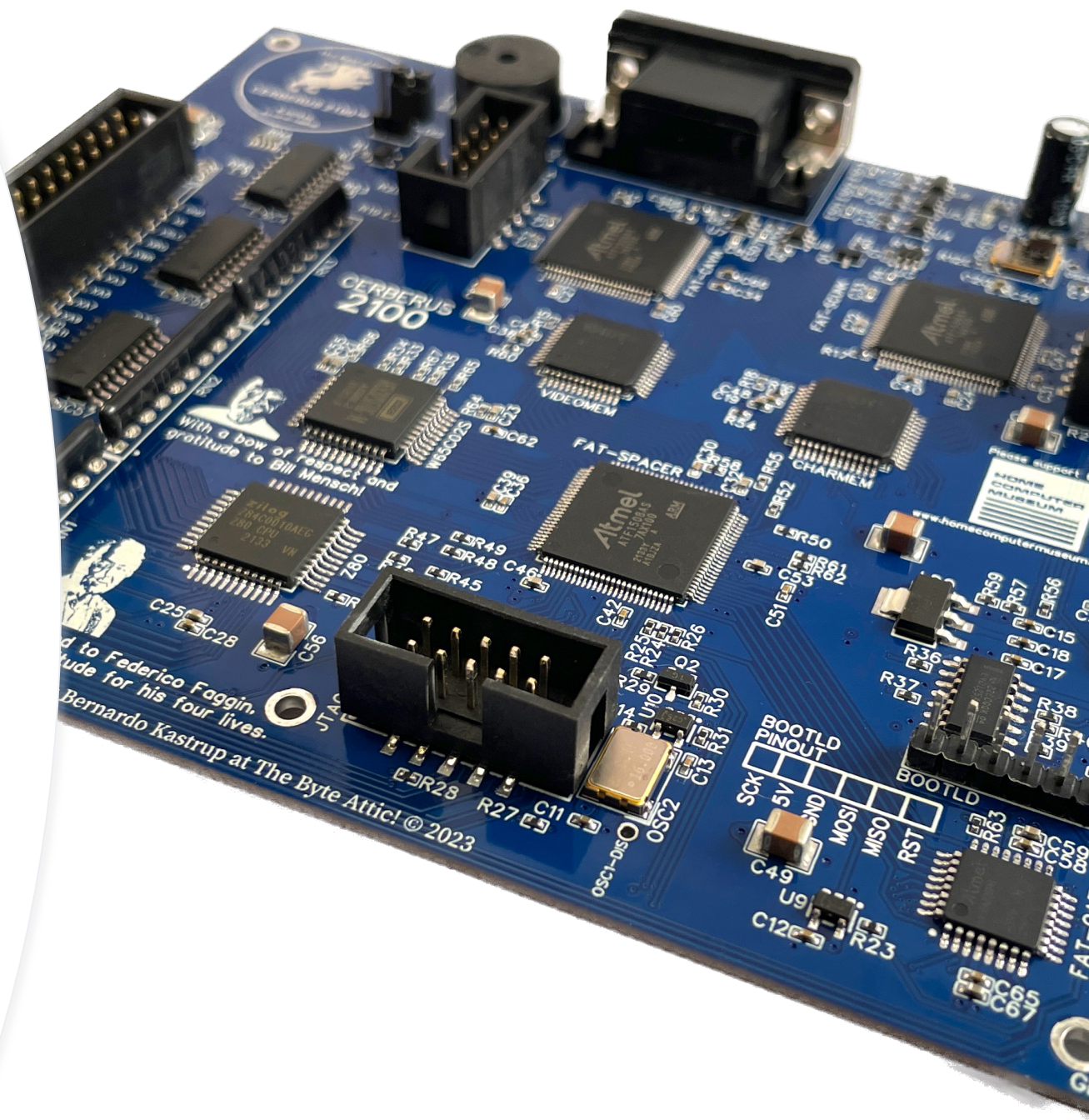
Tips and tricks

- You can load new programs by turning CERBERUS 2100™ off, removing the µSD card and copying new files to it from your PC
- Remember that you can control the CPU clock speed from the BIOS Screen: if a program is running too slow or—which is more likely—too fast, you can partly compensate for it with the **`slow`** and **`fast`** commands
- Clean the CERBERUS 2100™ board only with ESD-safe brushes and/or ESD-safe compressed gas
- There are tiny, modern PS/2-compatible USB keyboards—such as the *MC Saite*—which can be used with CERBERUS 2100™
- Monitor updates to the CERBERUS 2100™ distribution files for firmware improvements and eventual bug fixes
- Since FAT-SPACER, FAT-SCUNK and FAT-CAVIA are CPLDs, even hardware updates to the heart of CERBERUS 2100™ can be performed without physical modifications to the board

The Byte Attic's

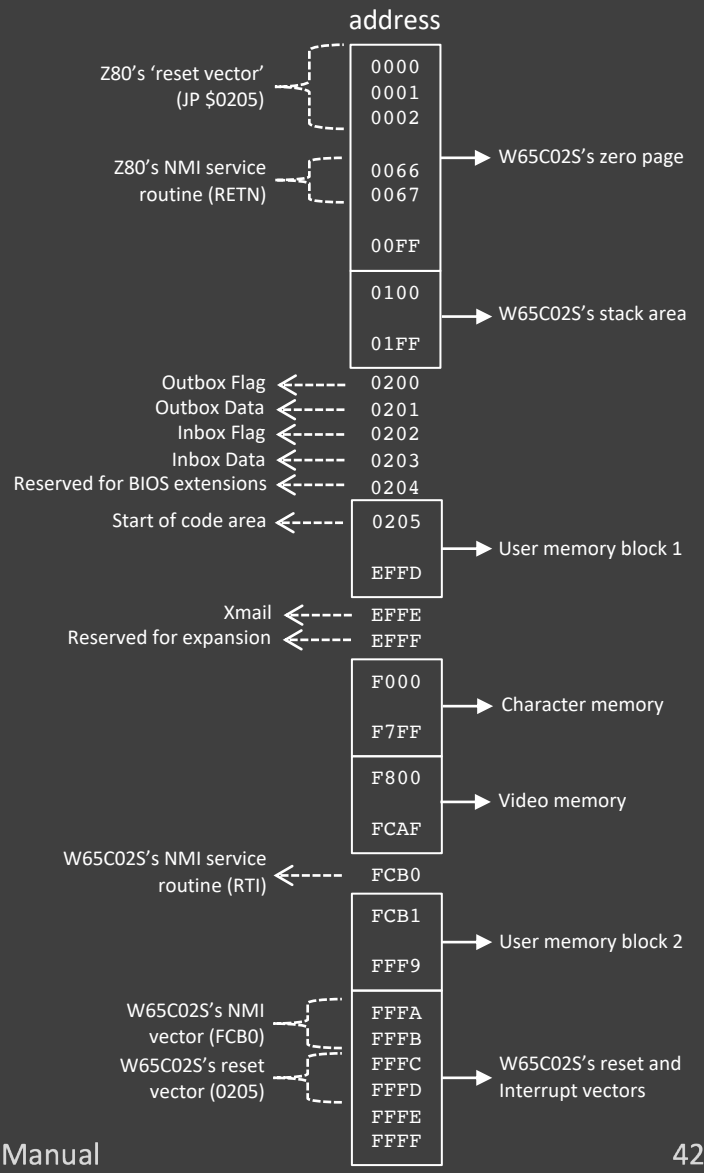
CERBERUS 2100™

Notes for programmers

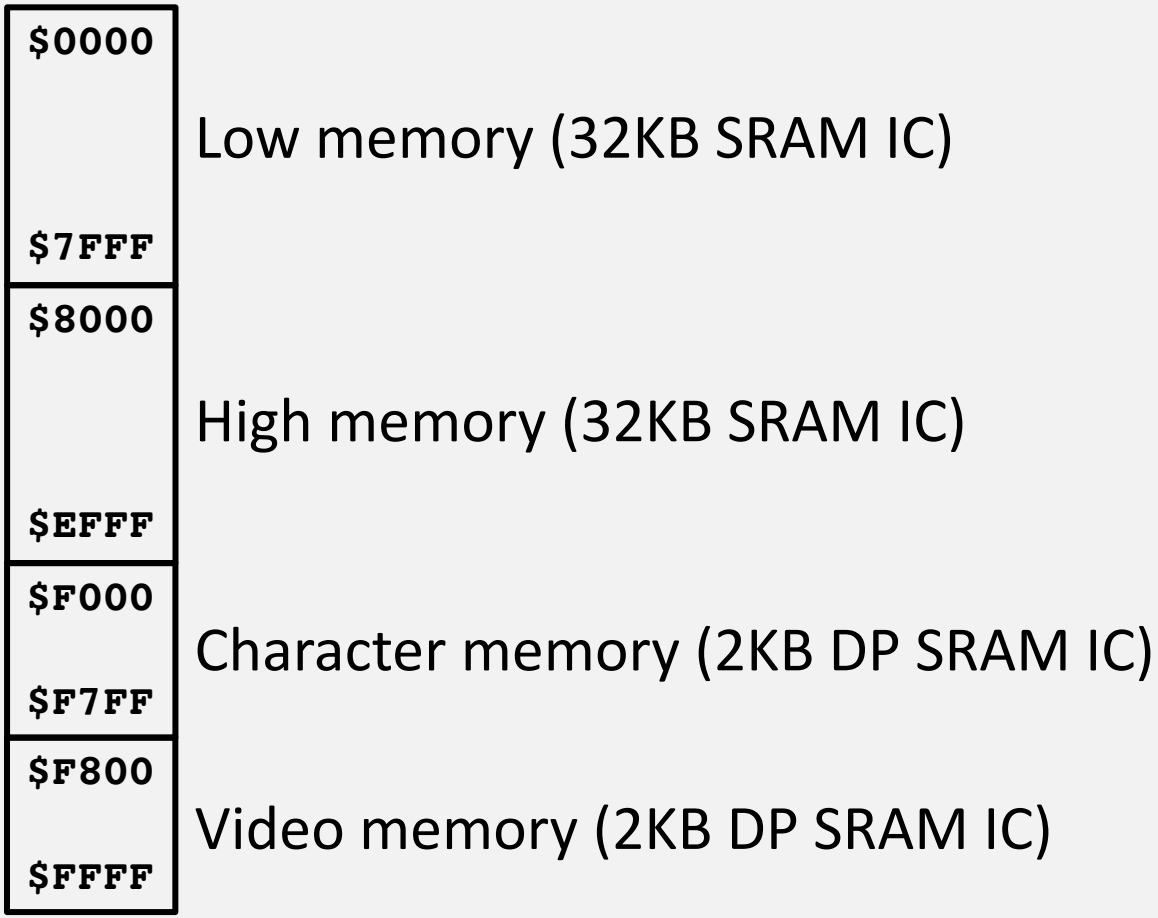


More memory map considerations

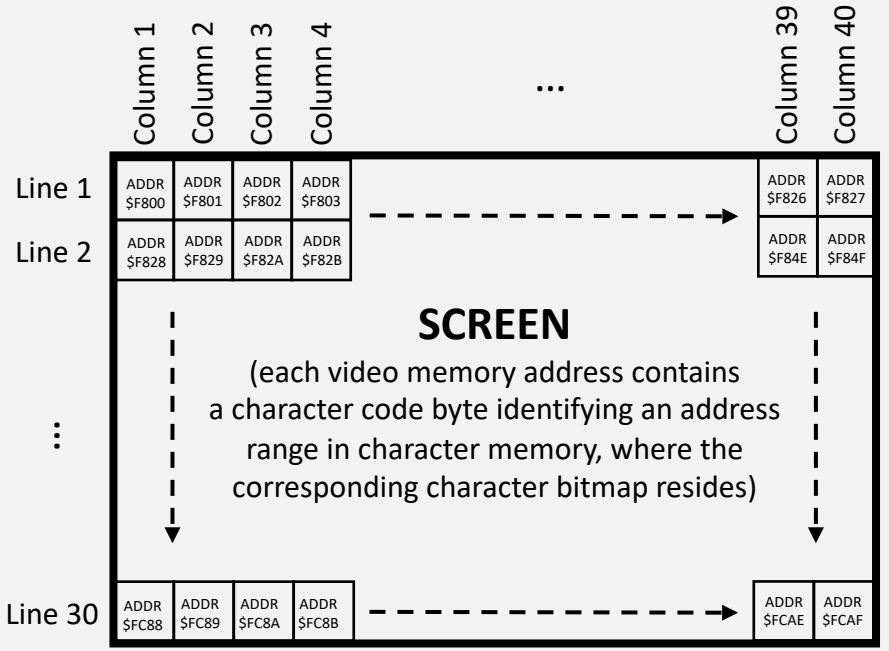
- The BIOS will automatically put a `JP $0205` on addresses `$0000`, `$0001`, and `$0002` to simulate a reset vector for the Z80
- The BIOS will automatically put a simple `RETN` (return) instruction on addresses $0066 and $0067, where the Z80 looks for its non-maskable interrupt service routine
- All I/O between FAT-CAT and CPU is memory-mapped to addresses `$0201` (Outbox from FAT-CAT to CPU) and `$0203` (Inbox to FAT-CAT from CPU)
- Address `$0205` is the start of the code area for both CPUs
- The BIOS will store an `RTI` (return from interrupt) on address `$FBC0`, which is the non-maskable interrupt service routine for the W65C02S
- The BIOS will ensure that the W65C02S's NMI and Reset vectors are set according to the above
- The largest contiguous area of memory available for code and variables is the user memory block 1
- User memory block 2, physically residing in the DP Video SRAM, is also available to programmers
- Z80 coders can use the address space of the W65C02S's zero page and stack area as regular memory (except for addresses `$0000` to `$0002`, `$0066` and `$0067`, which are reserved)
- Z80 coders can also use the top 6 addresses as regular memory (which are otherwise reserved for the W65C02S's reset and interrupt vectors)



Physical memory map



Video memory organization



More programming notes

- CERBERUS 2100™ supports two instruction sets: the Z80's and the W65C02S's
  - But applications must target one *or* the other CPU, not both concurrently
- FAT-CAT handles keyboard inputs when a CPU is running an application
  - Whenever a key is pressed, FAT-CAT halts the CPU, pokes a 1 into the Outbox Flag address, pokes the corresponding character code into the Outbox Data address, releases the CPU, and then issues a non-maskable interrupt (NMI)
- Applications should always clear the Outbox Flag by poking a 0 into it immediately upon reading out the Outbox Data
- The BIOS defines the non-maskable interrupt service routines as mere returns from interrupt (`RTI` for the W65C02S and `RETN` for the Z80)
  - An application running on a CPU must read keyboard inputs by polling the Outbox Flag to see if there is a new input from the keyboard, and then reading out the inputted character code from the Outbox Data address
- If you want to use more elaborate NMI service routines for keyboard input—instead of Outbox Flag polling—you will need to adapt the BIOS code (which is simple to do, since the BIOS is an Arduino sketch written in C)

More programming notes (cont.)

- Since FAT-CAT also issues a non-maskable interrupt upon updating the Outbox Data and Outbox Flag addresses, wait-for-interrupt instructions (`WAI` for the W65C02S, `HALT` for the Z80) are supported
- Maskable interrupts are *not* supported
  - Non-maskable interrupts are edge-triggered and, therefore, much simpler to handle in the hardware
  - CERBERUS 2100™ is an educational platform unlikely to be used for real-time applications, so the lack of maskable interrupts shouldn't be a problem
- The Z80's port instructions (such as `IN` and `OUT`) are *not* supported
  - All I/O is memory-mapped, through the Outbox and Inbox Data addresses
- The `.asm` and `.hex` files of the example applications are provided in CERBERUS 2100's distribution, which illustrate how to deal with Outbox Flag polling and wait-for-interrupt instructions in both CPUs
- I also recommend that programmers writing applications for CERBERUS 2100™ acquaint themselves with the BIOS code, available in the distribution

Generic cross-assembler considerations

- Your code should *always* start at address `$0205`, *even for the Z80*
  - The BIOS will put a `JP $0205` on address `$0` to simulate a reset vector
  - You should tell the assembler that your code starts at `$0205`
- CERBERUS 2100's loader is very simple: it merely reads the binary file's bytes and stores them sequentially in memory, from address `$0205` onwards
  - Only addresses and opcodes should be part of the assembled machine code!
  - No loader directives are supported

- See the files in the folder

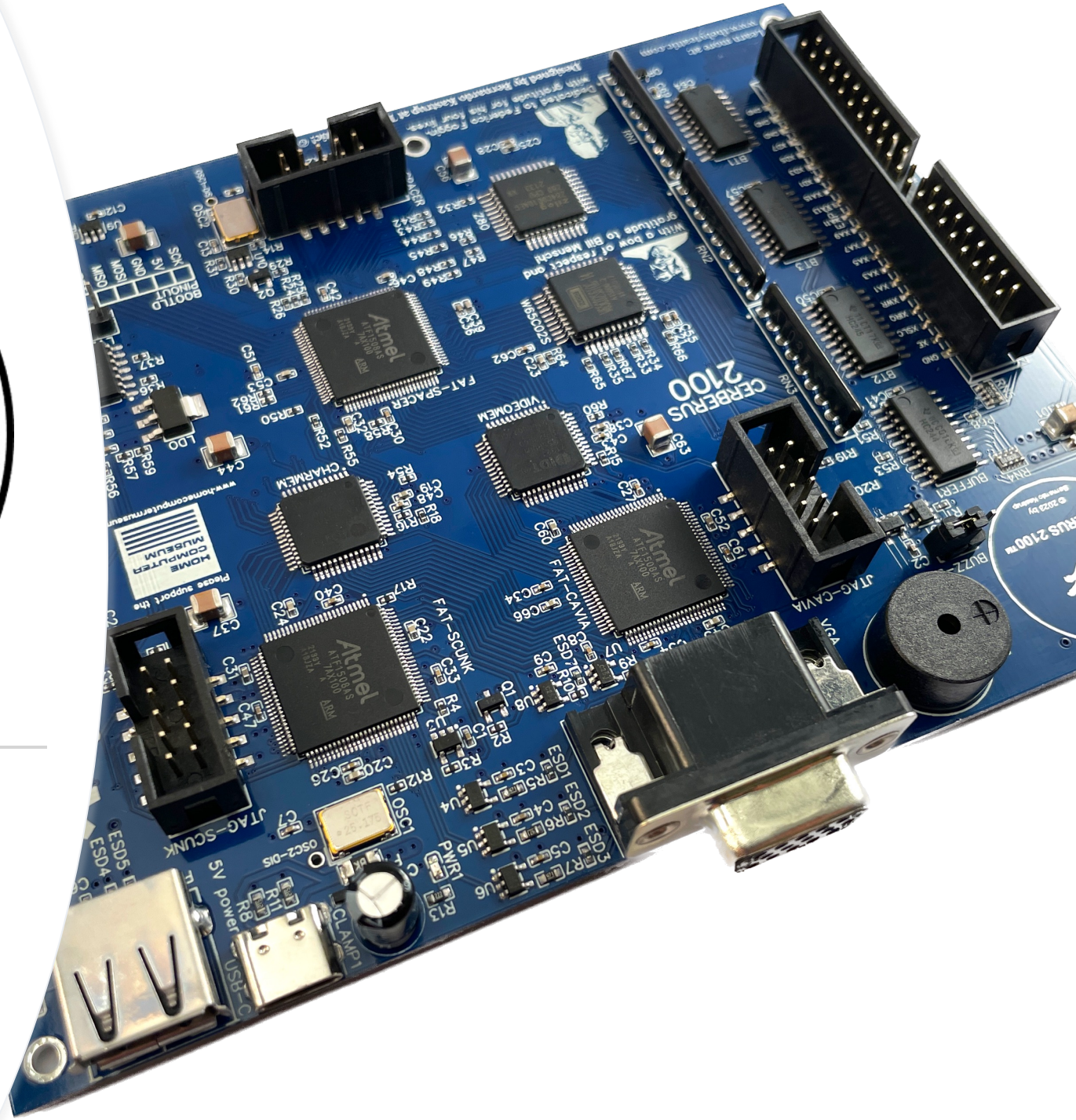
  `/CERBERUS Applications Source Code`

  for examples of assembly programs for both CPUs
- If you want to change character definitions as part of your code, you should do it with `load` (Z80) or `store` (W65C02S) assembly instructions, as CERBERUS 2100's loader will not recognize address directives

The Byte Attic's

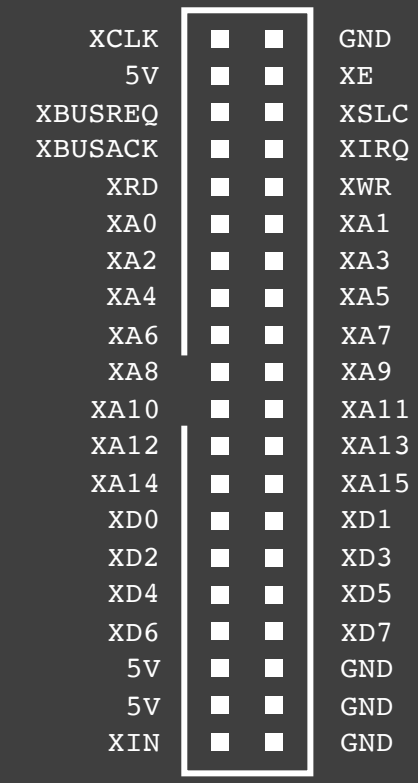
CERBERUS 2100™

Expansion slot



Expansion slot overview

- CERBERUS 2100™ has a 40-pin expansion slot
- Cards connected to the expansion slot can have Direct Memory Access (DMA) to all of the system's memories
  - Therefore, the cards can have CPUs and microcontrollers; they don't need to be passive-only circuits
- All communication between an expansion card and the rest of the system is memory-mapped
- The expansion I/O protocol is generic, so cards can work with either CPU
- The expansion slot's pinout is marked directly on the board's silkscreen, but is repeated here for convenient



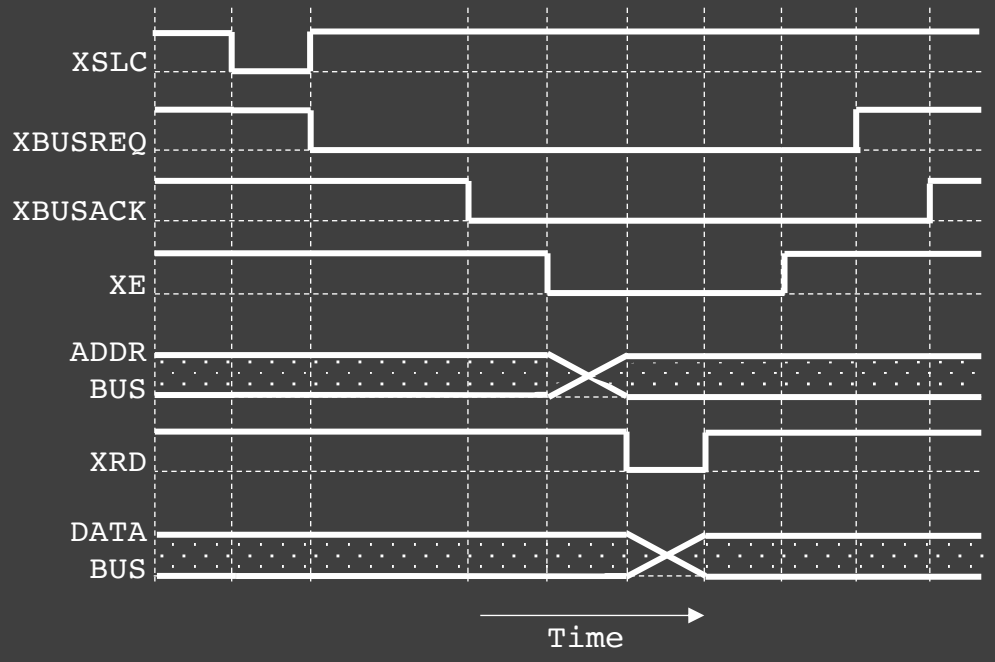
Expansion slot signals overview

- All control signals are *active-low*
- Inputs/Outputs
  - **XA0** to **XA15**: address bus lines, as exposed to the expansion slot through transceivers
  - **XD0** to **XD7**: data bus lines, as exposed to the expansion slot through a transceiver
- Inputs from the system, into the expansion:
  - **XCLK**: the CPU clock (4 or 8MHz, as selected by FAT-CAT)
  - **XBUSACK**: acknowledgment from FAT-CAT that the CPUs are tristated and the system's buses are now available to the expansion card
  - **XSLC**: a strobe from FAT-SPACER indicating that the *Xmail* address in high memory has been written to and, therefore, there is data in memory for the expansion card
- Outputs from the expansion, towards the system
  - **XBUSREQ**: indicates that the expansion card is requesting (DMA) access to the system
  - **XE**: enables the data and address bus transceivers so the expansion card can sniff the buses or perform DMA
  - **XIRQ**: interrupt strobe that signals to the rest of the system that the expansion card has put something in memory for it
  - **XRD**: memory read strobe from a card
  - **XWR**: memory write strobe from a card
  - **XIN**: unused output connected to FAT-SPACER, so the expansion I/O protocol can be extended by users

DMA read procedure & timing diagram

The expansion card should perform the following steps to read from system memory:

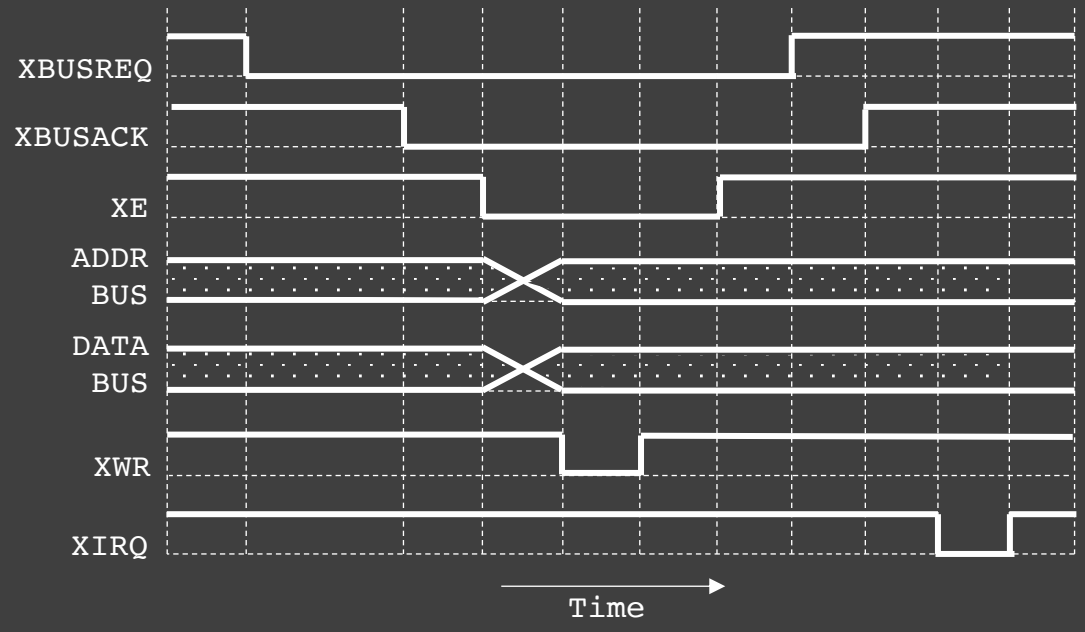
1. Wait for **XSLC** to pulse low, signaling there is data in memory for the expansion card
2. Request access by pulling **XBUSREQ** low
3. Set address value locally, in the card
4. Wait until **XBUSACK** goes low
5. Pull **XE** low to drive the system's address bus with the locally set address value
6. Wait for the system's address bus to stabilize
7. Pull **XRD** low to trigger memory output
8. Wait for the system's data bus (output from system memory) to stabilize
9. Read in the system's data bus contents
10. Push **XRD**, **XE** and **XBUSREQ** back high (in this order) to complete the read cycle



DMA write procedure & timing diagram

The expansion card should perform the following steps to write to system memory:

1. Request access by pulling **XBUSREQ** low
2. Set address and data values locally
3. Wait until **XBUSACK** goes low
4. Pull **XE** low to drive the system's address and data buses with the locally set values
5. Wait for the buses to stabilize
6. Pull **XWR** low to trigger memory write
7. Wait for memory write cycle to complete
8. Push **XWR**, **XE** and **XBUSREQ** back high (in this order) to complete the write cycle
9. Wait for **XBUSACK** to go back high
10. Pulse **XIRQ** low to indicate to the rest of the system that the expansion card has written something in system memory



Bus sniffing

- A card on the expansion slot can ‘sniff’ the contents of both address and data buses without requesting access through **XBUSREQ** or being granted such access through **XBUSACK**
- To do so, the card can simply pull **XE** low, so to enable the bus transceivers
  - There is no need for a read strobe **XRD**
- The CPU clock signal **XCLK** can then be used for synchronization, allowing the values on the data and address buses to be sampled when they are stable
- Sniffing mode allows for passive expansion cards that simply monitor CERBERUS 2100™’s internal bus traffic
- By monitoring when writes are done to video and character memory, it is conceivable that, for instance, alternative video cards could be built
  - Even though the read and write strobes within the system are not available in the expansion slots, access to video and character memory addresses from the computer proper are almost always write accesses; and even if they are read access, the data bus will still correctly reflect the contents of the respective address

Notes for card designers

- The expansion I/O protocol is asynchronous, so you are *not* required to use **XCLK**; your card can use its own crystal/oscillator, as long as you remain alert to the memories' access timings (if your card's clock is slower than 10MHz, you need not worry about it)
- All expansion control signals are already pulled up with 10KΩ, so there's no need to pull them up again
- The data & address buses are also pulled up with 10KΩ, *but only on the system's side of the bus transceivers;* therefore, your card should pull up its own internal data & address lines
- All signals are referenced to 5V at CMOS levels
- Remember that *all expansion control signals are active-low*
- **XSLC** is a *pulse*, so polling it won't work
  - You must connect it either to a pin of a microcontroller that has an edge-triggered interrupt attached to it…
  - …or to a set/reset flip-flop, which can in turn be cleared by your logic once the transition of **XSLC** has been handled; this way you can poll the state of the flip-flop to know if **XSLC** has gone low

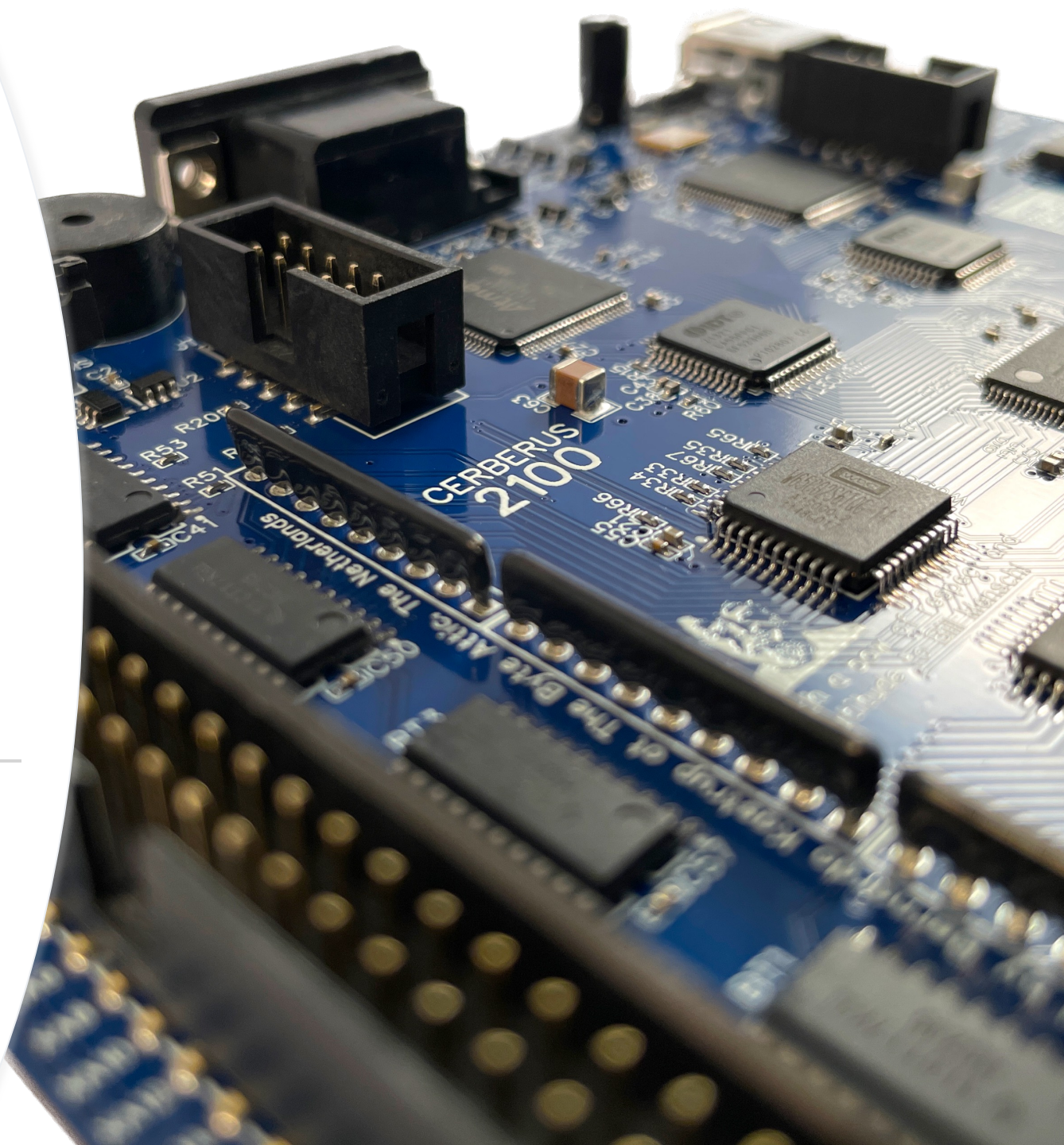
Notes for card programmers

- The top of high memory (address `$EFFF`) is reserved for the expansion, so feel free to use it as you wish
- All communication to and from the system should be mapped onto *the system's memories*
  - If your card has on-board memory, that local memory should *not* be accessed directly by a CPU, but only by the card itself
  - If your card has on-board sensors, then the results of their measurements should be written to system memory before they can be accessed by a CPU
- **XSLC** goes low when the *Xmail* address is written to, but that does *not* mean that only the byte written to *Xmail* can be communicated to the expansion card
  - As long as the card and the code running on a CPU are compatible, a protocol can be software-defined so the card reads an entire block of memory at each read cycle
  - A final write to *Xmail* by a CPU can then work as a trigger for that read cycle
  - The same rationale applies to the card's write cycles: they don't need to be restricted to a single byte

The Byte Attic's

CERBERUS 2100™

CPLD-based design



CPLD design notes

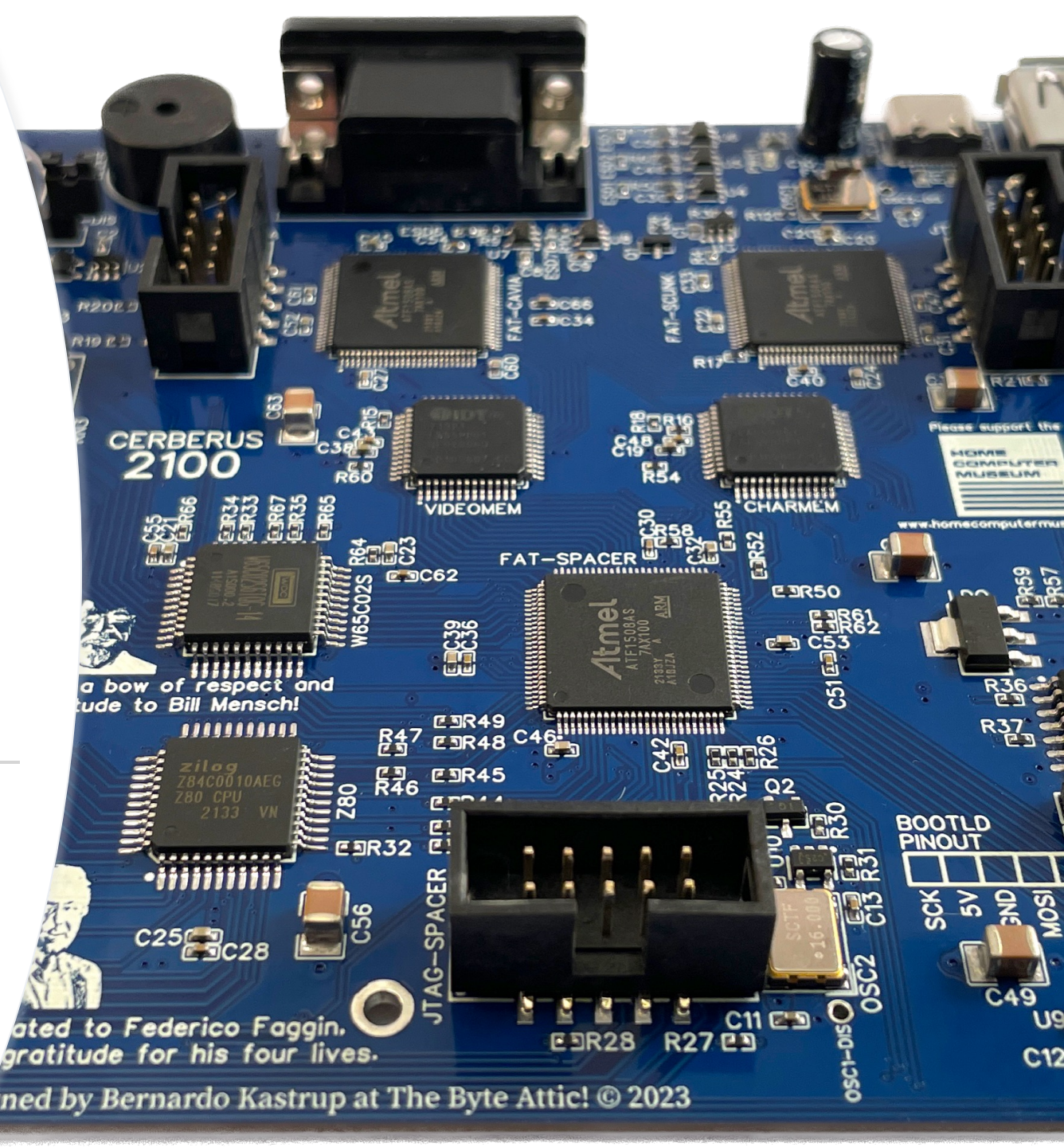
- These brief notes aim to merely *complement* the extensive commentary embedded in the `.PLD` files provided with CERBERUS 2100's distribution, which contain the full hardware description of the three custom ICs
- The CUPL hardware design language used is extensively described in the CUPL documentation available online
- The CUPL software is freely-available, so you can experiment with changing the design of the custom chips, if you so dare
- Beware: the CUPL software doesn't like empty spaces in any directory or file name, and will fail to find necessary files if these empty spaces are used
- The pinouts of FAT-SCUNK, FAT-CAVIA and FAT-SPACER can be seen in the respective `.fit` files in the distribution

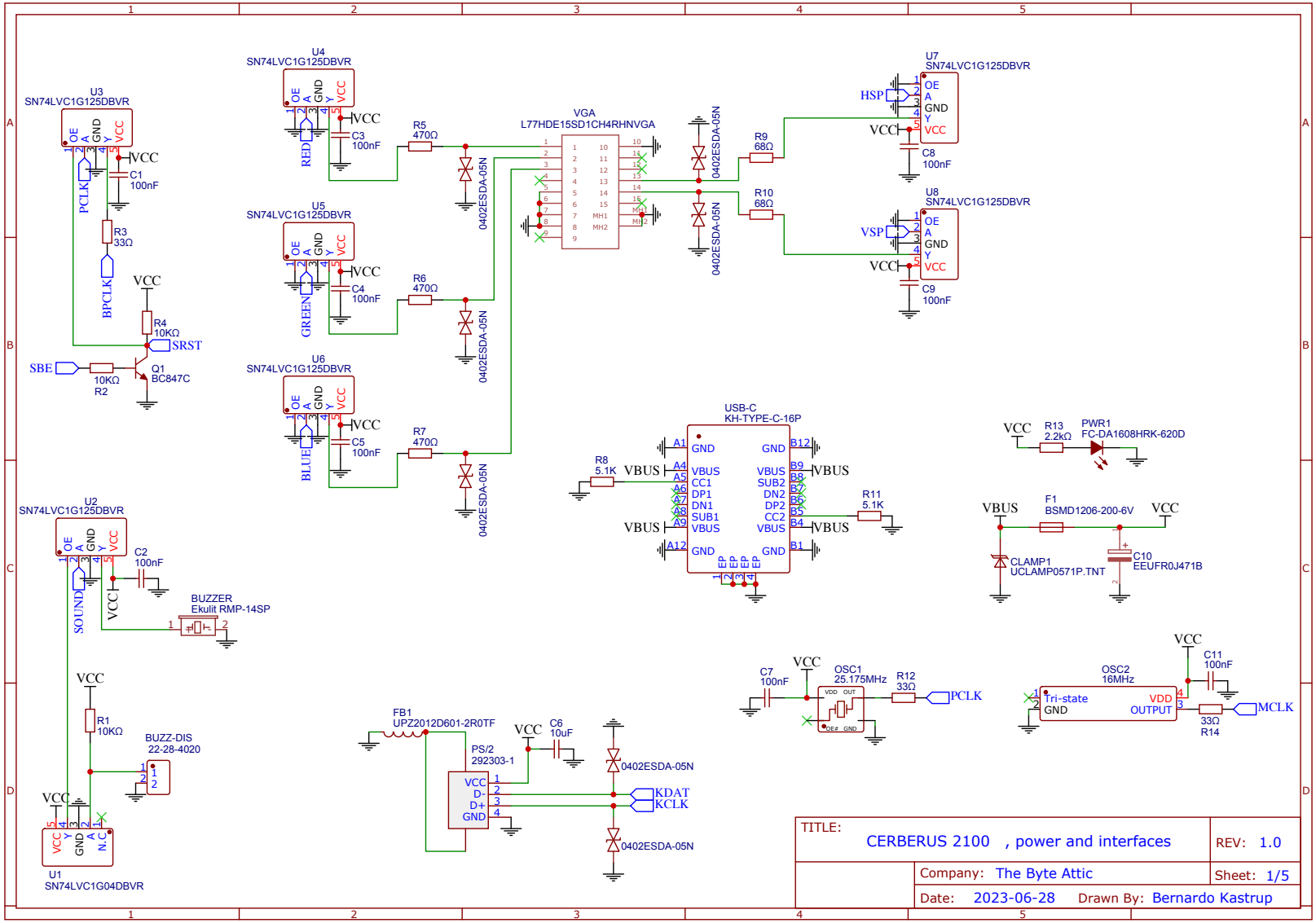
- Very briefly, here is the CUPL syntax:
  - '`&`' means a logical AND
  - '`#`' means a logical OR
  - '`!`' means a logical INVERSION
  - '`SIGNAL.t`' is the input to a toggle flip-flop whose output is `SIGNAL`
  - '`SIGNAL.d`' is the input to a D flip-flop whose output is `SIGNAL`
  - '`SIGNAL.ck`' is the clock input to a flip-flop whose output is `SIGNAL`
  - '`SIGNAL.ce`' is the clock enable input to a flip-flop whose output is `SIGNAL`
  - '`SIGNAL.ar`' is the asynchronous reset input to a flip-flop whose output is `SIGNAL`
  - '`SIGNAL.ap`' is the asynchronous preset input to a flip-flop whose output is `SIGNAL`

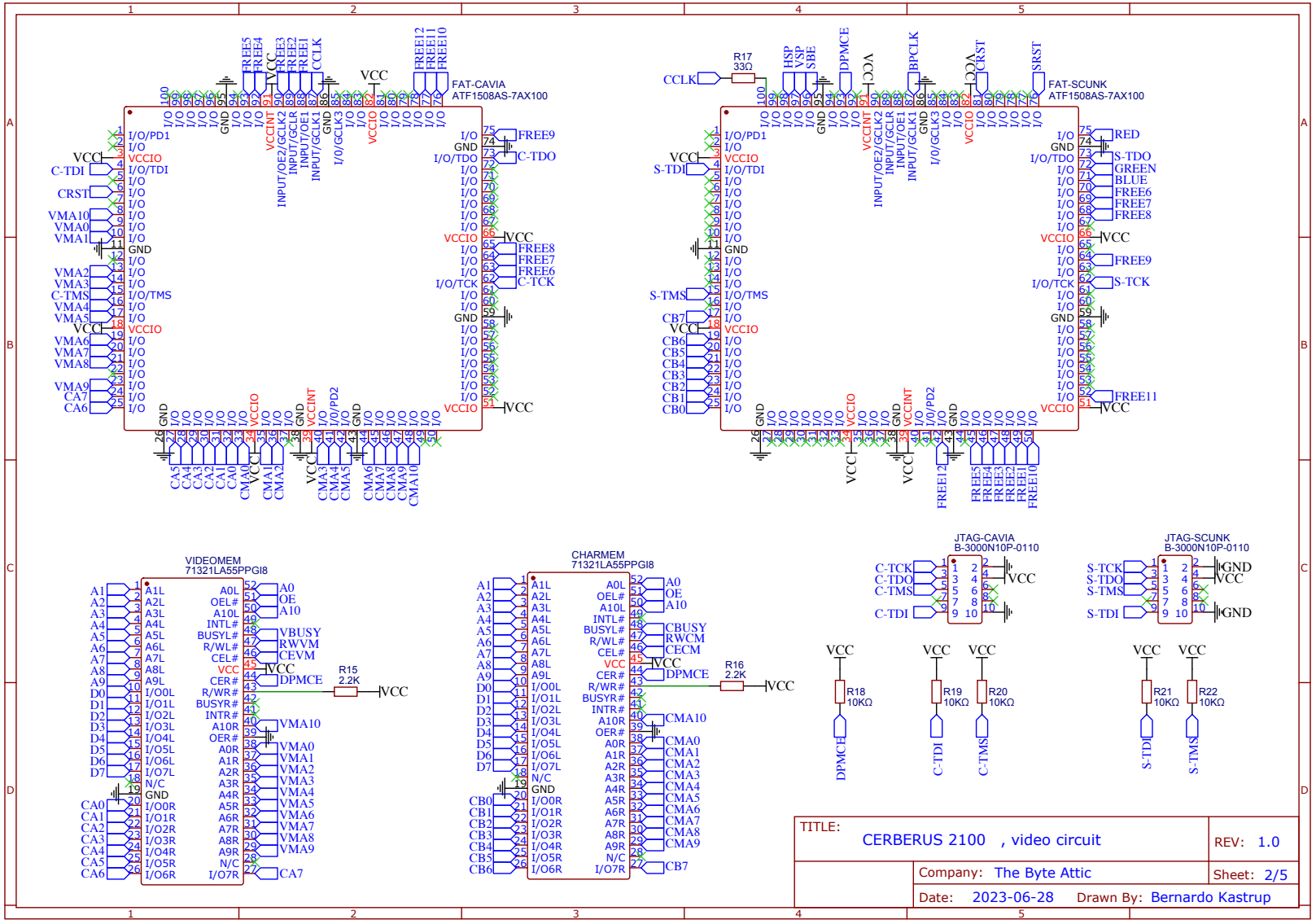
The Byte Attic's

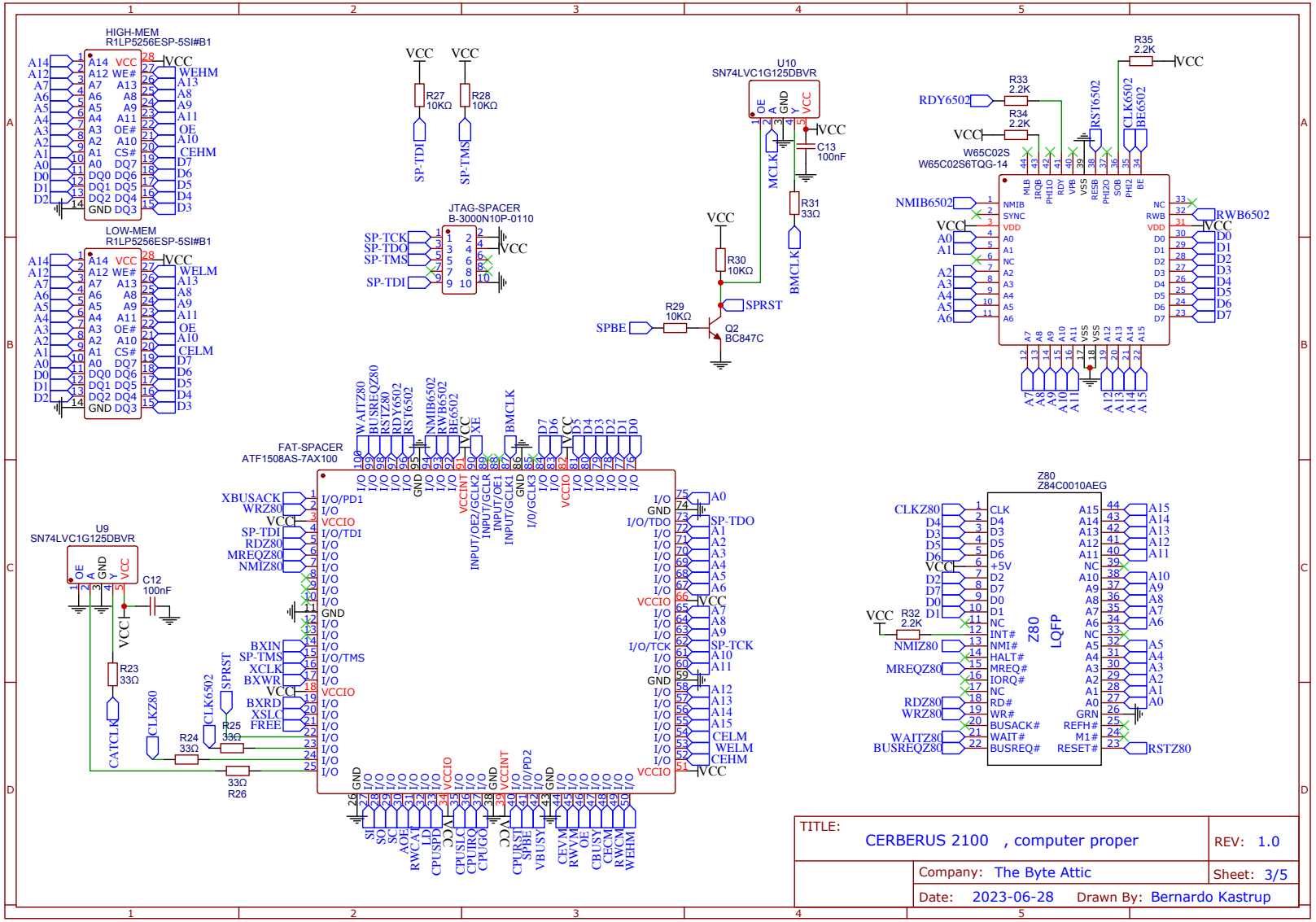
CERBERUS 2100™

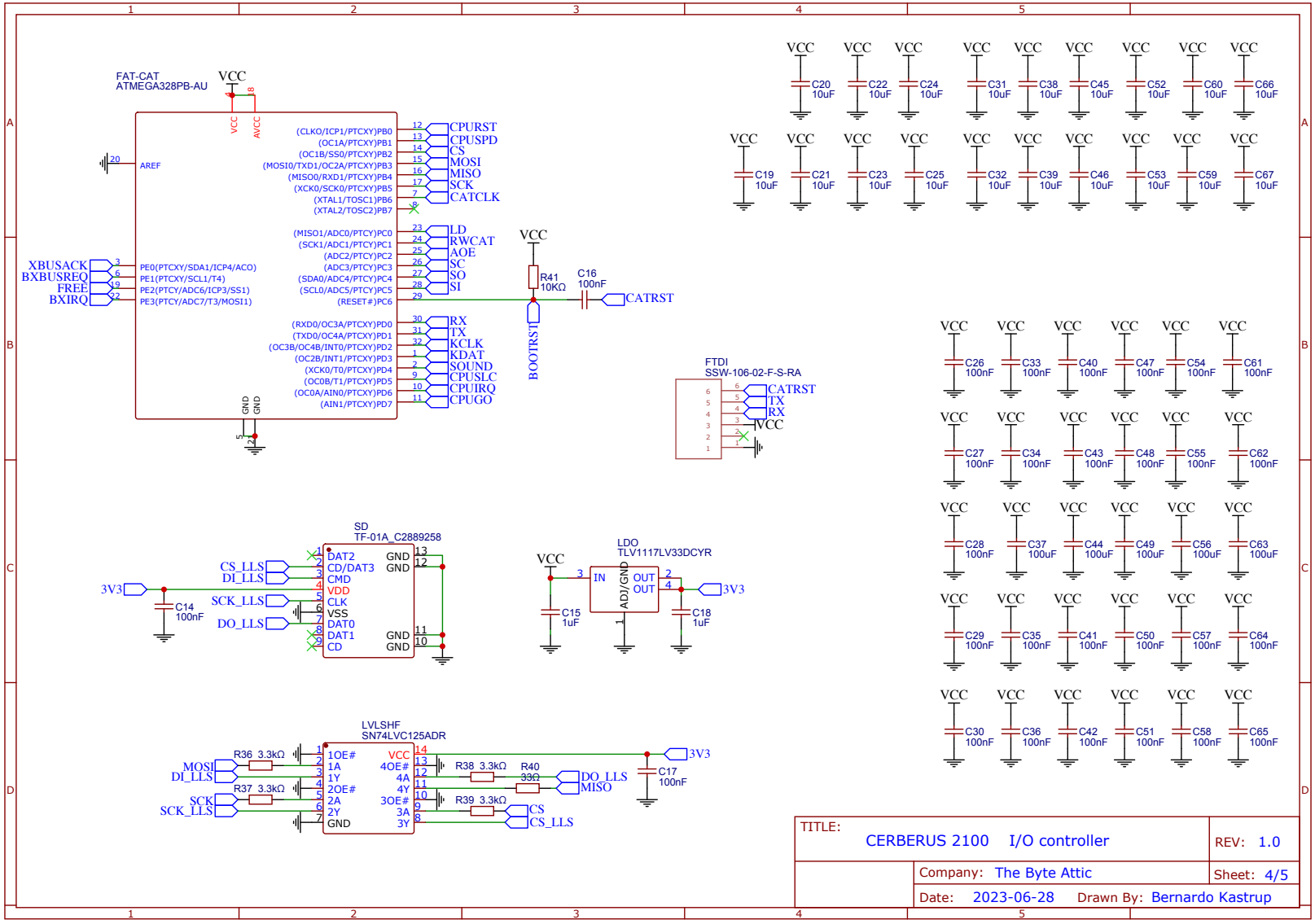
Schematics

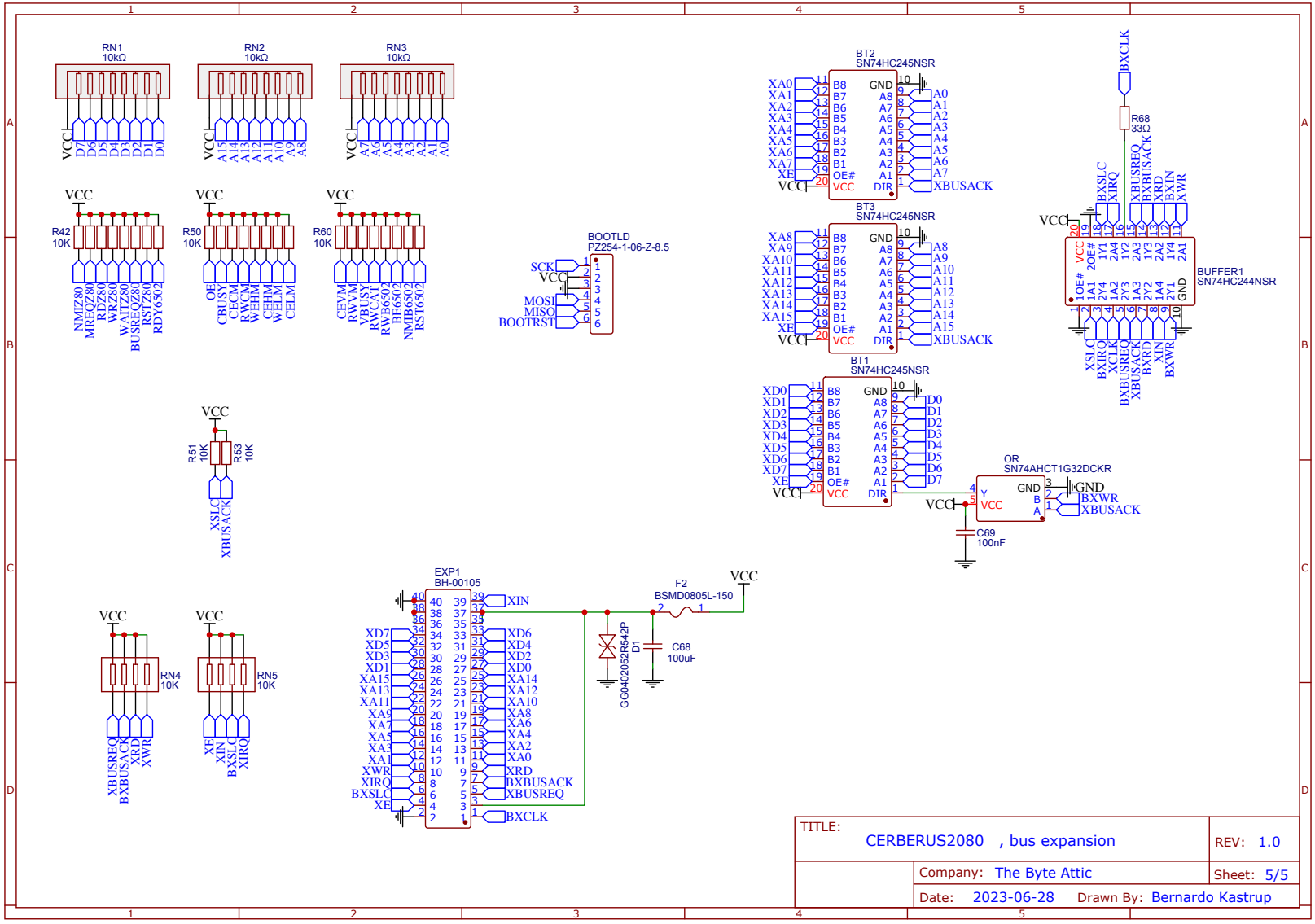



U3
SN74LVC1G125DBVR
PCLK
C1
100nF
R3
33Ω
BPCLK
R4
10KΩ
SRST
SBE
10KΩ
R2
Q1
BC847C
U4
SN74LVC1G125DBVR
RED
C3
100nF
R5
470Ω
0402ESDA-05N
U5
SN74LVC1G125DBVR
GREEN
C4
100nF
R6
470Ω
0402ESDA-05N
U6
SN74LVC1G125DBVR
BLUE
C5
100nF
R7
470Ω
0402ESDA-05N
VGA
L77HDE15SD1CH4RHNVGA
0402ESDA-05N
0402ESDA-05N
R9
68Ω
R10
68Ω
U7
SN74LVC1G125DBVR
HSP
C8
100nF
U8
SN74LVC1G125DBVR
VSP
C9
100nF
USB-C
KH-TYPE-C-16P
R8
5.1K
R11
5.1K
VCC
R13
2.2kΩ
PWR1
FC-DA1608HRK-620D
F1
BSMD1206-200-6V
CLAMP1
UCLAMP0571P.TNT
C10
EEUFR0J471B
U2
SN74LVC1G125DBVR
SOUND
C2
100nF
BUZZER
Ekulit RMP-14SP
R1
10KΩ
BUZZ-DIS
22-28-4020
U1
SN74LVC1G04DBVR
FB1
UPZ2012D601-2R0TF
PS/2
292303-1
C6
10uF
0402ESDA-05N
KDAT
KCLK
0402ESDA-05N
C7
100nF
OSC1
25.175MHz
R12
33Ω
PCLK
OSC2
16MHz
Tri-state
GND
VDD
OUTPUT
C11
100nF
33Ω
R14
MCLK
TITLE:
CERBERUS 2100 , power and interfaces
REV: 1.0
Company: The Byte Attic
Sheet: 1/5
Date: 2023-06-28
Drawn By: Bernardo Kastrup


FAT-CAVIA
ATF1508AS-7AX100
FAT-SCUNK
ATF1508AS-7AX100
VIDEOMEM
71321LA55PPGI8
CHARMEM
71321LA55PPGI8
JTAG-CAVIA
B-3000N10P-0110
JTAG-SCUNK
B-3000N10P-0110
R15
2.2K
R16
2.2K
R17
33Ω
R18
10KΩ
R19
10KΩ
R20
10KΩ
R21
10KΩ
R22
10KΩ
TITLE:
CERBERUS 2100 , video circuit
REV: 1.0
Company: The Byte Attic
Sheet: 2/5
Date: 2023-06-28
Drawn By: Bernardo Kastrup


HIGH-MEM
R1LP5256ESP-5SI#B1
LOW-MEM
R1LP5256ESP-5SI#B1
JTAG-SPACER
B-3000N10P-0110
U10
SN74LVC1G125DBVR
U9
SN74LVC1G125DBVR
FAT-SPACER
ATF1508AS-7AX100
Z80
Z84C0010AEG
Z80
LQFP
W65C02S
W65C02S6TQG-14
R27 10KΩ
R28 10KΩ
R30 10KΩ
R29 10KΩ
R31 33Ω
R33 2.2K
R34 2.2K
R35 2.2K
R32 2.2K
R23 33Ω
R24 33Ω
R25 33Ω
R26 33Ω
C12 100nF
C13 100nF
Q2 BC847C
SP-TDI
SP-TMS
SP-TCK
SP-TDO
SPRST
SPBE
MCLK
BMCLK
CATCLK
CLKZ80
CLK6502
RDY6502
RST6502
CLK6502
BE6502
NMIB6502
RWB6502
WAITZ80
BUSREQZ80
RSTZ80
NMIZ80
MREQZ80
RDZ80
WRZ80
XBUSACK
WRZ80
BXIN
BXWR
BXRD
XSLC
FREE
VCC
GND
TITLE:
CERBERUS 2100 , computer proper
REV: 1.0
Company: The Byte Attic
Sheet: 3/5
Date: 2023-06-28
Drawn By: Bernardo Kastrup


FAT-CAT
ATMEGA328PB-AU
VCC
AREF
XBUSACK
BXBUSREQ
FREE
BXIRQ
CPURST
CPUSPD
CS
MOSI
MISO
SCK
CATCLK
LD
RWCAT
AOE
SC
SO
SI
RX
TX
KCLK
KDAT
SOUND
CPUSLC
CPUIRQ
CPUGO
R41
10KΩ
C16
100nF
CATRST
BOOTRST
FTDI
SSW-106-02-F-S-RA
SD
TF-01A_C2889258
CS_LLS
DI_LLS
SCK_LLS
DO_LLS
3V3
C14
100nF
LDO
TLV1117LV33DCYR
C15
1uF
C18
1uF
LVLSHF
SN74LVC125ADR
R36 3.3kΩ
R37 3.3kΩ
R38 3.3kΩ
R39 3.3kΩ
R40
33Ω
C17
100nF
MOSI
DI_LLS
SCK
SCK_LLS
DO_LLS
MISO
CS
CS_LLS
C20 10uF
C22 10uF
C24 10uF
C31 10uF
C38 10uF
C45 10uF
C52 10uF
C60 10uF
C66 10uF
C19 10uF
C21 10uF
C23 10uF
C25 10uF
C32 10uF
C39 10uF
C46 10uF
C53 10uF
C59 10uF
C67 10uF
C26 100nF
C33 100nF
C40 100nF
C47 100nF
C54 100nF
C61 100nF
C27 100nF
C34 100nF
C43 100nF
C48 100nF
C55 100nF
C62 100nF
C28 100nF
C37 100nF
C44 100nF
C49 100nF
C56 100nF
C63 100nF
C29 100nF
C35 100nF
C41 100nF
C50 100nF
C57 100nF
C64 100nF
C30 100nF
C36 100nF
C42 100nF
C51 100nF
C58 100nF
C65 100nF
TITLE:
CERBERUS 2100 I/O controller
REV: 1.0
Company: The Byte Attic
Sheet: 4/5
Date: 2023-06-28
Drawn By: Bernardo Kastrup


RN1
10kΩ
RN2
10kΩ
RN3
10kΩ
BT2
SN74HC245NSR
BT3
SN74HC245NSR
BT1
SN74HC245NSR
BUFFER1
SN74HC244NSR
BOOTLD
PZ254-1-06-Z-8.5
R68
33Ω
R42
10K
R50
10K
R60
10K
R51
10K
R53
10K
RN4
10K
RN5
10K
EXP1
BH-00105
F2
BSMD0805L-150
C68
100uF
C69
100nF
OR
SN74AHCT1G32DCKR
TITLE:
CERBERUS2080 , bus expansion
REV: 1.0
Company: The Byte Attic
Sheet: 5/5
Date: 2023-06-28
Drawn By: Bernardo Kastrup