HB63B09M Prerelease Programmers Reference

© D. Collins – 2025

Intro and Forward:

The HB63C09M was an idea I hatched back two years ago when I was finishing up my first home-brew computer the HB6809. But it goes back further than that. The beginning is really around 2020 when I started building kit computers as a means to pass the time during the pandemic. I had purchased a kit from McJohn in Italy called the Z80-MBC2, by Fabio Defabis – an excellent little computer kit with a single logic chip, atmega32, Z80 CPU and a RAM chip. I was hooked on the small form factor and the things that I could do with my little computer. I made a few hardware expansions for it, most notably a joystick interface for my Tetris clone that I had programmed in pascal. At the time there were people asking Fabio on the Facebook group for the project if he was going to do other processors. Before long he was working on the V20-MBC (which I built) and later he went on to make the 68K-MBC which is architecturally slightly different. After a while, no new MBC computers were made, and by then I was on to building my 6809 based system, and still quite a novice at this (I am still!)

Things happened life got busy I spent some time working on other projects and about a year ago started playing around with Arduino – and digging very deeply into the Z80-MBC2. I really wanted to understand how memory was loaded using a microcontroller and I wanted to know how Fabio had shared the bus with the Z80. Right around the same time I had also stumbled onto an even older work, Brad Rodreguez's "Scroungemaster" and I fell in love. The scroungemaster is (well was) a concept for a 6809 multi-processing build Brad had built on wire wrap. He had used a rather simple circuit to do clock stretching in order to hand off cleanly to the other CPU's into common ram. He detailed this in a paper entitled "Multiprocessing for the Impoverished" (found at https://www.bradrodriguez.com/papers/) I quietly started to play around with the design and build a small expansion for the HB6809 that allowed me to use an Arduino like a peripheral without a latch simply by stretching the clock.

In his paper, Brad discusses the parallels between the Z80 and the 6809. I found a lot of parallels in my Arduino sketch and the work that Fabio had done with the Z80-MBC2, and after a few months I had a working prototype that could stage itself from the Arduino (at the time using latches) and boot into a machine language monitor. After that I streamlined the design, making it slightly more like the Z80-MBC2's little brother the Z80-MBC by opening up a few address lines to the Arduino and then additionally solving a very difficult timing problem by changing from a controller generated clock to a synchronous clock (both the 63C09's internal clock and the ATMEGA32 are clocked by the same 20Mhz crystal) and the board fully stages from ROM files saved to the SD card. No Rom code exists on the AVR EEPROM (save a small less than 40 byte program that bootstraps from the SD card). This allows the programmer and user to switch between environments without having to Flash a ROM onto the computer!

After a few revisions of the board, I found I could further shrink the form factor to a smaller 10x10 PCB – albeit 4 layers. The Exact same size PCB as the Z80-MBC2! For me, it was like coming full circle, and now I think I am ready to share this design with others. What I have is a fully functioning – ready to go 63C09 playground. But there's still some work on the software, this is where you come in.

What I am calling this board revision is "The Prerelease" edition. Essentially the hardware is finished. From this point, any and all architecture changes will need to be approved by the "Community" -- what that means specifically will have to depend on the group of developers that go ahead and early adopt the design for the express purpose of porting software to it. For now there is enough (really) for people to start playing around with the platform in earnest, basic, machine language monitor are all there. I have started this document which is a rough outline of a manual for

programmers and developers. But to really go forward I need somebody who's more versed in operating systems than I am. I could possibly figure out how to continue developing software and add LBA device support to basic and possibly the monitor – but I would very much be learning as I go. So The main purpose of this release is to get the boards out there in to peoples hands, hopefully people with considerably more systems knowledge who have a passion for the 6x09 CPU and can really take the lead on developing things like FLEX, Nitros09, Forth and hopefully a lot more!

What follows in section 0 gets you through assembling and programming the board if yours is not. And then in Section 1, we go over the Architecture in a high level general overview format, what I think is enough information to really go on if you want to start developing for the HB63C09M. This section also contains a breakdown of all the functions of the Atmega32, as well as generally how the expansion connector works.   Then in Section 2, we go over the memory map It is fairly short and not much more than a table but contains everything you need to know of the stock memory layout of the board. In section 3 we go over the staging environment as well as briefly over how iOS expects LBA images to be set up on the SD card.   What follows is a simple appendix of the comments and remarks from the Arduino sketch, these explain how each function works and how to properly send data for each function.

An appendix B was added to include pin-outs for the connectors on the board.

I assume that you know your way around a computer, that you understand how to configure a serial terminal inside Linux or windows. I am assuming Linux/OSX environment here, so it is important to remember if something does not make sense or you are confused to please reach out on FB, Discord or put a Issue report in Github. If all else fails please email me at z80dad@gmail.com.

Additionally – if somebody wants to take the reigns on this document – I will gladly step back to anybody who feels that they can do better, I am generally OK at writing documentation but its not my favorite, and I can't say what the quality is like for other people.

Dedication:

Thank you to Fabio Defabis, who's work on his MBC single board computers allowed for the this design to come to be.

And To Grant Searle, who's minimal CPU designs keep inspiring new and creative projects

Also Brad Rodriguez, for writing what is the definitive starting place for 6809 multiprocessing

But also to Jeff Tranter, Dave Dunfeild, DigicoolThings, and Jonny Quest.

Also to Arthur Collins – For is insights on ringing, engineering life, fatherhood and more. You continue to make all things possible for me.

Art Collins – "You need to print it out, get out a pencil, and just go over them one at a time."

Contents:

Section 0 – PCB Assembly and Programming the Atmega32:

The PCB is a 10x10 4 layer board with a full edge to edge Power and ground plane's. Keep this in mind when soldering the components to the board. The most important thing to remember is that the Ground and VCC pins on each chip and socket will be very difficult if not impossible to extract without cutting the legs on the chips or sockets. So go slowly and deliberately checking your work as you go.   You will need flux, a decent soldering iron, solder, side cut pliers and small tweezers or a pair of needle nose pliers for holding things in place while soldering. Some nice things to have are a good flux cleaner (I use Berryman electronics cleaner, after dissolving the flux with isopropyl alcohol). I assume some general electronics knowledge is known by the assembler. Therefore I won't go into specifically how to solder, use flux and clean the tip on your iron. There are many good videos on the internet explaining exactly this as well. Obviously the most basic disclaimer with retro / hobbyist computing – there are no safety nets here – everything has a modicum of risk and everything is 100% at your own risk if you choose to build this project.

Generally speaking you would like to solder the parts in the following order:
- resistors and diodes soldering the standing resistor R6 last (this is the E Clock pin resistor)
  - you can lay R6 Down sideways like this:
- followed by the small bypass capacitors (these are not polarized and go in any way – they are all the same value)
- then soldering U11, U4 and U3 (these are the two transceivers and the bank register). These must be soldered without a socket, completely flat to the board. Very likely if you make a mistake here it will be very hard to check as these chips are underneath the two 40 Pin DIP packages and can not be placed in a socket.
- Solder the tiny pad jumper on the board that is labeled "please close for normal operation" if it is not already closed on your board. (this should be fixed on the newer boards but some of the initial run boards that I sent to developers wanting to help require this step). The current recommendation is to take a resistor leg and lay it across the pads with a little dab of flux to hold it in place long enough to join the two pads.
- Resistor packs next – Please pay attention to the specific pin out and value of each resistor pack. Soldering these in the wrong way or using the wrong value will result in a non-working board.
- Depending on if you want the edge connectors for the serial chip and the sd card module to lay flat you may want to solder these before doing the next step. Future expansions may rely on these being sideways laying down on the side of the board so it might be a good idea to do it this way.
- Next sockets you would like to use. You can socket all of the chips but technically only have to socket the CPU and the Atmega32. I highly recommend installing a socket for the the very sensitive RAM chip.   I don't recommend turned pin machined sockets for the CPU, controller or the ram chip as many of these come with additional cross pieces and may not sit tall enough for the chips under the main IC packages to completely clear the bottom of the installed 40 pin packages. Instead use a good High quality "double wipe" spring socket without a cross piece.   You can carefully trim a cross piece from the center of the socket so that it neatly fits over U11, U4 and U3.
- Lastly solder the remaining taller parts in any order that suits you. Once all of your sockets are soldered use a multimeter to check for bridged pins and shorts between pins, especially power and ground. Use the schematics to verify pins that are supposed to be soldered together. Do not forget to close JMP1 if you are going to use the USB dongle for power

Note: to get the SD card interface to fit with larger standoffs it is some times required to trim the screw hole corner on the PCB with tiny side cutters into a square so that the connector will seat properly

The atmega32 is an integral part of the chip set for the hb63C09, and handles all of the IO processes. Without programming this chip there is no way to stage or boot the computer. And while this is not difficult, it is an important step which must be completed after soldering the PCB together. For the time being a small ISP programmer (or another Arduino) is required to fash the atmega32's programming to the chip. There are many guides on the internet for using an Arduino as a programmer, so here I will lay out using a USBtiny programmer:

- You will need a USBTiny programmer like this one from Amazon (or similar):
  - https://www.amazon.com/HiLetgo-USBTiny-USBtinyISP-Programmer-Bootloader/dp/B01CZVZ1XM
  - TinyASP is also verified to work with an 8 to 6 pin adapter.
- You will need a FTDI or TTL to USB adapter with DTR or RTS. The pin out is configured for the FTDI FT232RL adapter like this one:
  - https://www.amazon.com/HiLetgo-FT232RL-Converter-Adapter-Breakout/dp/B00IJXZQ7C
- You will need Arduino, freely available from:
  - https://www.arduino.cc/en/software (I prefer the legacy version but the newer one should work too)
  - you may also have to set up "dial out" to make programmers and serial to usb converters work under linux. Directions for this can be found here: https://support.arduino.cc/hc/en-us/articles/360016495679-Fix-port-access-on-Linux
- You will need to install Mighty core:
  - https://github.com/MCUdude/MightyCore (directions for this in the repository)

After installing / obtaining the above you will need to select the chip This can be found under:

Tools → Board → MightyCore → Atmega32

And then Selecting the clock speed of your crystal oscillator:

Tools → Clock → External 20Mhz

Lastly Check your programmer is selected, for USBTinyISP pick:

Tools → Programmer → USBTinyISP (slow)
you may try "fast" however, I have noticed that the "slow" selection causes fewer issues over all.

Once the environment is set up, you will need to flash the bootloader onto the chip. This is a requirement for timing, clock and chip settings to be correct and so that you can upload with the serial bootloader, instead of the ISP programmer.

 First start by removing the SD card, and your USB to serial interfaces from the PCB top port near the SD activity light. And then plugging the 6 pin programmer ribbon cable which came with your programmer into the HB63C09, and then into your programmer. Lastly you can plug the USB

connection into your PC.   If you've properly installed everything the Power LED should turn on on the HB63C09. **If you do not remove the SD interface, prior to programming** the controller you could damage the reset pin / programming circuits inside the micro controller which could result in the chip not being able to be upgraded from the chips UART pins or worse, so please remember this step.

To actually flash the boot loader, select **Tools → Burn Bootloader**, (you should have already selected the programmer, USBTinyISP in the last step). This will take less than 10 seconds and should say successful in the log read out at the bottom. You may have to pick a (slow) option when picking the programmer if you are having issues with the boot loader in the next step, or with programming the controller boot loader, at all.

Next you need to open the Arduino sketch and program it to the controller. You do this by opening the file: **<location of repository>/HB63C09/Arduino/HB63C09M/HB63C09M.ino.** Next remove the ISP programmer ribbon cable, and with the SD interface still removed, install the TTL to USB interface.

Don't forget to close JMP1 by the USB interface if you are powering the computer from the USB, and don't power the computer from multiple headers, as this could cause damage. The AUX port, ICSP and USB to TTL port can all power the board, and the +5V is denoted on each of them with a small triangle arrow pointer "head".

Select the USB to TTL interface's serial port under **Tools → Port → <port for USB to TTL>.** Under Linux this is typically */dev/ttlUSB0*, though it may be another device completely this will depend on multiple factors including other hardware and configuration of your system.

Click **Sketch → Upload** (or press ctrl - u)

your board should be loaded once this completes, as the sketch is quite large the upload should take a good few moments to complete

You should prepare your SD card now by copping the contents of **<location of repository>HB63C09/System Roms/BIOS.BIN** to the root of the SD Card, or extracting the SD Card ZIP file from the same directory

To test, you will need to load a serial terminal program such as *minicom* or *putty*. (both are free)

The Serial settings for your terminal are **115200kBps , 8 N 1** (eight data bits, no parity and 1 stop bit). **No hardware / software flow control**. The interface will typically be at **/dev/ttyUSB0** under Linux.

To properly install minicom on linux (Debian / Ubuntu) from a terminal type:
sudo apt-get install minicom

To enable color if desired edit .bashrc in your home directory and add the lines

#minicom color
MINICOM='-con'
export MINICOM
(you will need to log out and back in to enable this)

To set your system wide serial port:
sudo minicom -s

Then arrow down to select **Serial Port Setup <enter> → A (this should automatically move you to the field)**

Once in the field, erase the entry and type the name of your serial port on linux this is usually **/dev/ttyUSB0 <ENTER>.**

At this screen you can toggle hardware flow control, you should switch it off by typing F to toggle the hardware flow control. Hardware flow control needs to be off.

Then you can hit the **<ESC>** key to jump back to the main menu.

Then you need to arrow down to **Save setup as dfl <enter>** and arrow down to exit. If your USB to TTL cable is not plugged in this will error back to the bash prompt. If not, it should automatically reset the HB63C09 if its plugged in and start the boot up process described below (*if you have configured color in the previous step, it will not work until you exit and re-enter minicom without sudo rights.)*

To Exit minicom you type CTRL-A, Q <enter>

For help type CTRL-A, Z (which should bring up the context help)

If the computer boots it should drop you into the vBIOS environment, you will need to set the default settings which are **SET START C0000 <enter> SET SIZE 4000 <enter> SET FILE BIOS.BIN <enter>.**

Look these over with **SHOW <enter>** and save changes with **COMMIT <enter>**, and then type **QUIT <enter>.**

The computer should load into the MON9 machine language monitor if everything worked correctly.   You can switch to basic by typing **JB <enter>**, to leave basic and return to the monitor type **BYE <enter>**.

Section 1 – Architecture Overview

The chip set is designed to be as simple as possible, and so this introduces a few gotchas which we will go over here. The main chip besides the 63C09 CPU is the Atmega32 (hereafter referred to as the IO Controller), this is supported by seven other off the shelf logic chips. The data port on the IO controller is buffered by a bus transceiver to assure the controller is off the bus at the appropriate time. There is an external bank register and buffer which handle the banking, along with a few decoder chips, buffers, logic gates and inverters. These together create what is called the "Bella" chip set. This 7 chip set is planed to be condensed into a single CPLD / GAL chip for miniaturization which will be called "Isabella".

Clock stretching via MRDY:
 All IO exchanges introduce a clock stretch asserted by the MRDY signal. This means that any memory read or write to A000-AFFF (see *table a* below) will stretch the clock for a time. The length of time depends on how long the IO controller needs to complete the requested task.   As of this edition, specific timing of each IO operation is not fully documented, in the next edition a full timing diagram will be made available. For the time being its best to assume that the wait state is never longer than the data sheet maximum (5uS) though some are considerably faster (such as a mezzanine read / write). The IO controller releases the clock stretch with an ~IO_Grant signal which originates at a pin on the IO controller.

Halt / Reset / DMA:
The Halt signal, and the Reset signals are open collector inputs on the CPU which will be pull different signals to tri-state when asserted through buffers within the chip set.   Both signals have a pull up resistor which pulls their state to high if nothing is pulling them low (typically the IO controller).

When system reset is asserted, the MRDY line will be forced high by its pull up as it is cut off by the tri-state buffer when the active low reset pin is low. This effectively inhibits bus requests during reset and prevents logic on the mezzanine from causing an IO request during reset. Also keep in mind that the IO controller resets the CPU during boot up. The switch for reset is not the same reset line as the CPU's reset line. The Switch simply resets the IO controller which in turn resets the CPU. The Reset line which is available on the mezzanine is the CPU reset line, and will not reset the IO Controller.

When the halt signal is asserted, the ~BRD / ~BWR signals are tri-stated. While the CPU is in halt all of its busses (control, address and data) are tri-state. Since the intel similar RD / WR signals are generated by the chip set these signals must also be sent to tri-state during halt so that whatever is halting the CPU can strobe operations into the ram via these signals. NOTE: the R//W line and E are connected to the input of a 2 to 4 decoder, which generate the unbuffered RD/WR signals, during a halt these unbuffered signals are not reliable but in no way interface with the RAM chip while the halt signal is asserted. This also means that driving the R//W signal from an expansion during halt has no effect on the RAM chip. Conversely, a DMA device will only be able to to use R//W to access the ram chip, and will have to synchronize these transfers the same way the CPU would using the E clock (which is still generated during DMA).

IO Operations carried out by the IO Controller (bus mastering):
The IO controller is an ATMEGA32 based AVR. It can act as a very limited bus master, as well as handle IO requests like a peripheral IC. During boot the IO Controller halts the CPU and holds reset in order to shut down any IO requests and tri-state the bus. While in this mode it has access to

only the first 6 address lines A0-A5. Additionally, the address bus is tied high by strong pull ups which hold the bus at address FFFF. The CPU can use these address lines to write up to 63 bytes to the top of memory. This is just enough to load a small bootstrap program, as well as update the reset vector table. The first thing the computer does at boot up after asserting reset and halt is test to see if the SD card and RAM are available.   Once the status of the system is verified the IO controller strobes a very short program at the top of memory that contains the 63C09 for loading data off of the memory card. After this is loaded, the controller is set up to accept commands as a peripheral, and the halt and reset states are lifted. Once lifted the 63C09 jumps to the reset vector (the beginning of the self staging code), and runs it.

The self staging code relocates itself to 4000 hex, this location was chosen because it is the beginning of the fixed block and should work reliably even if the bank register is not functional. The code changes a few bytes of it's programming to point itself at the loader. After it is finished copying and self modifying, the 63C09 jumps to 4000 hex and starts loading the data from the card using the information from a few variables stored in the EEPROM of the controller, these variables are for start, size and name of the rom and are set with the vBIOS environment (refer to the section on vBios for operation here).   As long as the uploaded code properly sets the new reset vector, the computer resets and should boot to the staged code at the end of the copying process. The controller ultimately tracks the PC along with the CPU and sends the reset once the code is done coping. This is obviously a general overview of the self staging process. Please refer to the Sketch code comments as well as the source code for the loader for specifics into how the loader works. Additional information about the limitations of the loader can be found in the section on vBIOS.

*IO Operations carried out by the IO Controller (*peripheral*):*

In addition to staging the ram for boot up, the controller can also act as a Motorola type peripheral IC. Once the system has booted in 63C09 Mode the IO controller takes on a more traditional roll. It offers the ability to customize in software. At this time by modifying the Arduino sketch, however mini device drivers are planned. This will allow the 63C09 side of the computer to load AVR machine code into the EEPROM, which the controller will be able to integrate into the system on boot up for things like custom I2C devices and more – this may take some time to build as there are other priorities such as new hardware expansions for off board IO, video and keyboard etc…

To access the IO controller the programmer has to simply write to, or read from a memory mapped device IO address. A handful of options are set up at the moment but more are being thought of and added as time allows. Since we've reached what I consider a architectural milestone – there should be an expectation that nothing in the IO range should move or be changed in the main source fork without a community backed decision. So early development should not need excessive patching to resolve firmware updates.

Below is *table a1 and a2* containing a breakdown of the available programmed IO which can be used currently:

tables on next page

WRITE OPERATIONS:

ADDRESS	NAME	DESCRIPTION
0xA000	NULL	Legacy 6850 UART Control register.
0xA001	TXSERIAL	6850 Wrapper: Send a byte to TX Buffer (default buffer size is 64bytes)
0xA002	SELDISK*	Select Disk Number ie: 0..99
0xA003	SELTRACK*	Select Disk Track ie: 0..511 (two bytes in sequence)
0xA004	SELSECT*	Select Disk Sector ie: 0..31
0xA005	WRITESECT*	Write a Sector to the Disk (512 bytes in sequence)
0xA006		
...		*Future Use*
0xA03A		
0xA03B	TIMERCNT	*Timer Control Resister*
0xA03C	SETTICK*	*Set the Systick Timer a value of 1 to 255 msec*
0xA03E	LOADERR	This is the loader register it is unlocked by writing 255 after boot up.
0xA03F	SETBANK	Write the low nibble (3 bits) on the data bus to the bank register

Figure a1

READ OPERATIONS:

| ADDRESS	NAME	DESCRIPTION
0xA000	UARTSTAT	6850 Wrapper: simulates a 6850 Status register, see appendix a for details
0xA001	RXSERIAL	6850 Wrapper: Read a byte from the RX buffer (default buffer size is 64bytes)
0xA002	ERRDISK*	Read out the last disk error
0xA003	READSECT*	Read a sector from the disk (512 bytes in sequence)
0xA004	SDMOUNT*	Mount the installed volume (output error code as read value)
0xA006		
...		*Future use*
0xA03A		
0xA03C	TIMRSTA	*Timer Status Register*
0xA03C	SAMPTICK	*Sample the systick timer, a value between 0 and SETTICK* *
0xA03E	LOADER	This is the loader port its typically locked to the user.
0xA03F	RDBANK	Read the last selected bank value

Figure a2

* - These functions imported / ported from the Z80-MBC2 project by Fabio Defabis, an open source GPL licensed project. Please see attribution in repository, thank you Fabio.

Expansion Connector (Mezzanine connection):

 The Mezzanine connector pin out (F*igure 1*) shows the pin out configuration for the large expansion connector on the edge of the board. The expansion connection has no specific address decoding within the chip set on the main single board computer, instead – the expansion needs to provide at the point of connection:

- Buffers for bus and address connections.
- Address decoding within the range of 0xA040 and 0xAFFF.
- An active low, **open collector** request signal to !XSIN on the expansion header.

Figure 1
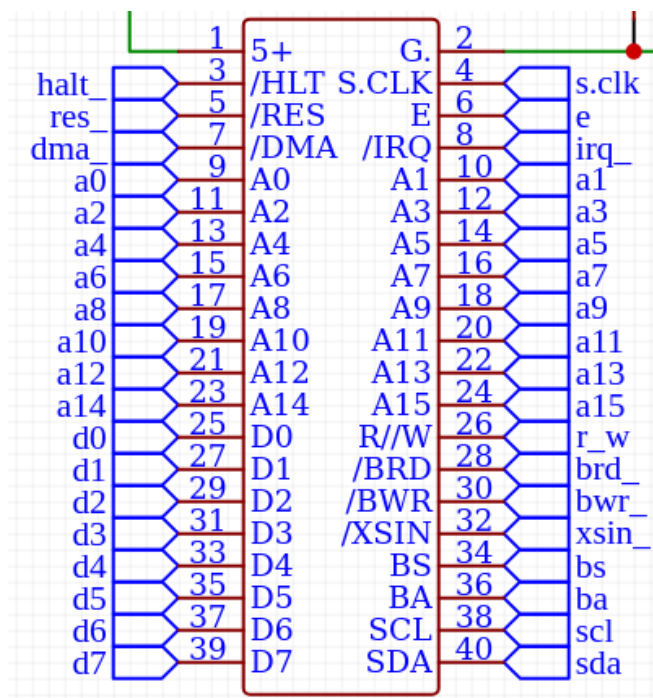
The !XSIN signal being brought low via an open collector output will trigger the controller to ignore any io request for the current cycle, as well as block the controller transceiver from enabling on the SBC.   At relatively the same time that the request comes in, the controller should see the IO request is from the Mezzanine and pass the grant signal along to the clock stretching circuit. This tiny wait state is not measured as of yet but should be by the next release of the document. For the time being please use the cpu address bus decay as the end of the cycle for driving the data bus, by driving your bus transceiver enable line to your address decode logic.

Remember – even though expansions are connected to the bus, the IO controller still needs to grant access to the mezzanine before the clock can continue. Additionally – any IO request within the range of A000 and AFFF will trigger an IO request.

The 74ALS156 is an example of a suitable decoder chip for driving !XSIN, another practical solution would be to use a single gate in a 74AHC125, following traditional push / pull logic by connecting the enable pin of the gate to the output, connecting the gate's input to ground and connecting the output to the !XSIN signal. In this configuration the output of the gate will ether be high Z (when the enable pin is high) or Low (when the enable pin is low). **Caution** connecting push pull logic to this pin could damage the inverter which drives the IO request logic.

Keep in mind, you can also re-map internal IO Controller functions by decoding the same address and sending the !XSIN signal. For instance – you could over map the on circuit UART with another controller or video circuit for that matter. By setting up address decode logic to pick of A000 and A001. This allows an upgrade pathway for core features, an is an unintended "feature" of the simplicity of the architecture.

Open Collector Requirements:

Open collector logic is a way to allow multiple outputs to be connected to a single input, with less expense and hardware. However it does require you follow some rules when deciding on logic types you would like to use in a perspective enhancement or expansion. Below is a list of all of the CPU / IO Controller connected signals that require this attention to detail:

- !HALT
- !RESET
- !IRQ
- !DMA
- !XSIN

use of any of these pins from the mezzanine, requires an open collector output of some sort. Driving these inputs high – even for a short while, could cause damage to these subsequent inputs. Please reference the perspective data-sheets for the HD63C09 and the ATMEGA32 on the requirements of these inputs as well. For the most part we have made reasonable accommodations, however there are some edge cases which require additional research in order to properly terminate links on any line.

I2C (work in progress) / SPI and UART:
I2C has been brought to the edge connector, however use of any device on the edge connector will require the software side to be configured inside the IO Controller sketch. I will work on an example case statement to show how this can properly be done using the wire library.

The SCL and SDA lines are tied high with 4.7k ohm pull up resistors which seemed like a good general purpose choice for the unterminated lines. Please keep this in mind when using I2C from the expansion and design your board to board connection appropriately.   SPI is exposed to the SD-Card interface, and to the ICSP port. As the HB63C09M uses the same architecture and circuit for SD cards as the Z80-MBC2 and others, you can not use both interfaces at the same time. This limitation was imposed to simplify the design.

UART is wired only to the Serial connection header, !DTR is also wired to the header and is required to do serial programming via the bootloader. In theory you should also be able to use !RTS at this pin as well, though untested.

Bank Register:
The bank register is how the computer knows which 16K page to expose to the bottom of the 64K memory map. The computer uses decode logic to determine if it should use the address bus transceiver or the bank register to drive the top most address lines on the 128K RAM chip. The value of the register is stored in two locations.   Firstly on the register itself, which is clocked by the IO controller during the write operation as its inputs are on the bus. Second, it is saved internal to the IO Controller, so that the user is able to read out the last bank number that was latched into the register. Under normal circumstances these two numbers should be the same, but its important to keep in mind the IO Controller has no way to read out the bank address, in the event that there is an issue with the bank register or the address transceiver, as these two values are stored in two different locations. To set the bank page you **write the bank number to memory location A03F.**   To read the last value written to the bank register **read from memory location A03F.** See table a1 and a2 for details.

Section 2 – Memory Map Overview

ROM / RAM Configuration

Addreses	64K Main Address Space
C000-FFFF	Upper Fixed RAM 16K Legacy Physical ROM Space
B000-BFFF	Upper Fixed RAM 4K Expanded Legacy ROM Space
A000-AFFF	Device Address Space for MCU 4K
8000-9FFF	Lower Fixed 8K of Fixed Lower Ram Area *
4000-7FFF	Lower Fixed 16K of Fixed Lower Ram Area *
0000-3FFF	Memory Bank Window, set by bank register (16K) *

128K RAM Chip Layout

1C000-1FFFF	Bank 7 (16K)
18000-1BFFF	Bank 6 (16K)
14000-17FFF	Bank 5 (16K)
10000-13FFF	Bank 4 (16K)
C000-FFFF	Bank 3 (16K) Upper Fixed RAM 16K Legacy Physical ROM Space
B000-BFFF	Bank 2 (4K) Fixed RAM Expanded ROM Space "Shadow" (3000-3FFF in bank)**
A000-AFFF	Bank 2 (4K) Bank 2 Free RAM Area "Behind" IO Space (2000-2FFF in bank)**
8000-9FFF	Bank 2 (8K) Lower Ram Area Top 8K "Shadow" (0000-1FFF in bank)**
4000-7FFF	Bank 1 (16K) Fixed Lower Ram Area Bottom 16K
0000-3FFF	Bank 0 (16K) bottom 16K of ram chip. Boots with this selected.

table b

* - when combined with the lower bank this totals 40K of contiguous memory
** - caution, there is shadowed ram in bank two between 0000-1FFF, and again in 3000-3FFF. Therefore the only "Usable Space" in bank 2 is 2000-2FFF These spaces are mapped again for clarity below:

Bank 2 Address Range	Description	128K Address Range
0000-1FFF	Lower Fixed RAM "Shadow" 8K	8000-9FFF
2000-2FFF	Usable Space 4K	A000-AFFF
3000-3FFF	Expanded ROM "Shadow" 4K	B000-BFFF

table c

On boot up the system resets the bank register, which selects bank 0. With this bank selected in a traditional 64K layout the system has access to 40K before the IO range (see *table b*), and 20K after the IO Range. All of these areas are RAM and can be written to as the ROM is loaded to RAM from SD card on boot up. Bank 2 overlaps both the Fixed memory areas (*table c),* a programmer should be careful to not accidentally overwrite data in these two locations as there are no safeguards within the very minimal chip set to prevent this from happening.

Accessing the IO range will initiate an IO request within the chip set. This causes the E strobe to stop for a time through the assertion of the MRDY signal. The IO controller will read or write from the bus and return control to the internal clock by sending a !IO_GRANT. This happens for ALL IO requests including any requests from the mezzanine connection, however in this case the IO controller releases the bus quickly as possible, however there is some small wait state when reading or writing to expansions. Exact lengths of all the wait states are still being determined, though many are less than a few micro-seconds, within the next release - full timing diagrams should be available for the IO controller. Please see the architecture overview for a complete listing of internal commands handled by the IO Controller.
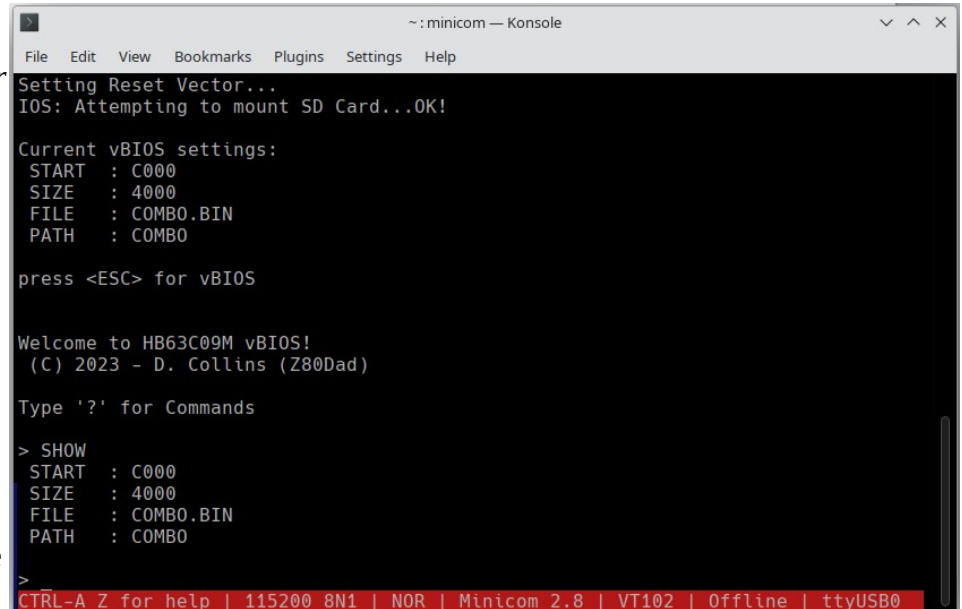
Section 3 – vCMOS Staging environment

The Staging environment is loaded by pressing <esc> when directed during a 3 second pause on boot up. This loads a very minimal dos like environment for setting the staging environment settings. Typing SHOW will bring up the list of system variables that can by set by the vBIOS environment:

Figure 2
(default staged environment for BIOS.ROM will look slightly different)

Each command assists in the setting of the environment variables: START, SIZE, FILE and PATH.

The START variable contains the start address for the loader.

The SIZE variable is how many bytes to be loaded before resetting the CPU.
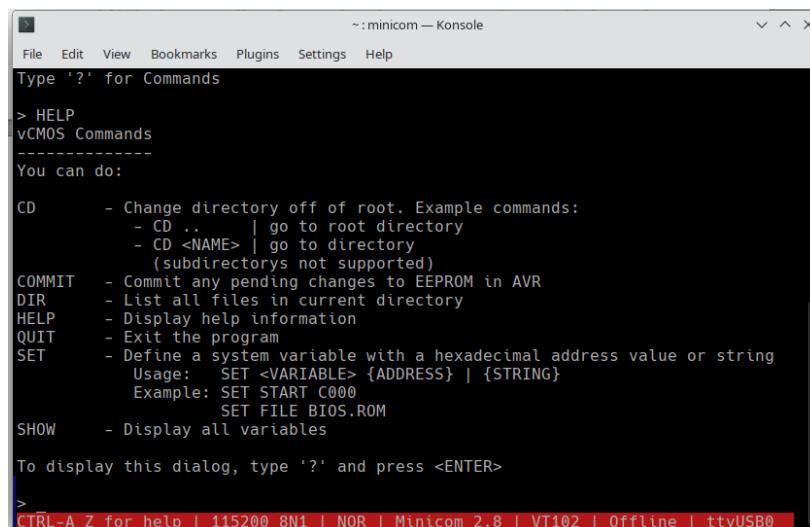
The FILE variable is what file to load within the selected PATH. vBIOS can only load a single file starting at a specific address. This file should ideally load to the end of memory so that it can overwrite the reset vector as the last operation the IO Controller performs is a reset to boot the system.

The PATH variable is the name of the directory to use when loading the environment. This variable is also used by iOS to determine where to load file system images from for any given system ROM

These variables are set by the specific commands which are inside the vBIOS Environment. You can type HELP <ENTER> or simply '?' <ENTER> to bring up the list of commands:

Figure 3

CD – switches directories within the root of the SD card, vBIOS supports switching to directories off the root, but not subdirectories. This is a limitation of how petitFS library works combined with wanting to keep the vBIOS environment as lean as possible. Additionally, vBIOS has no way of deleting or removing files from FAT due to the same limitations. – this is not a limitation of iOS however, as this uses file system images which can be manipulated exactly like a LBA device.

COMMIT – places the variable changes into the EEPROM inside the IO controller to be recalled at boot up.

DIR – brings up a list of all the files in the currently selected directory. Keep in mind it will list subdirectories if they exist but due to the way that path storage works, CD can only select directories off the root of the SD card.

HELP – Simply brings up the on line help message, the same result happens by sending the ? Character as well.

QUIT – Exits and attempts to boot the staged changes.

SET – sets a variable to a value see the examples pictured above. This can be a string value or a hexadecimal number – due to the way SET works, do not make the name of the rom file a Hexadecimal number, as this will potentially confuse the selection logic. Additionally setting to a system rom contained in a subdirectory will also update the PATH variable to the given path. Also note that you can not currently specify a path and a file separated by /. You must set the file and the path separately (or alternatively update FILE while in a subdirectory, which sets PATH). And update those values with COMMIT to change them.

SHOW – Shows the currently set variables used by boot up, keep in mind these variables still need to be saved to EEPROM with COMMIT if you wish to change what the computer uses at start up.

Limits of the loader:

The loader can only load files to ram which are contiguous. Additionally when the loader reaches the end of the file it resets the 63C09 CPU which causes a jump to the physical address stored at the reset vector. If the file loaded by the loader does not write a new reset vector, the computer will jump to whatever is stored there (by default this is the beginning of the bootstrap code which is at FFC0). If the loader bootstrap code was also not overwritten this will cause an infinite loop with the code loading to memory and the CPU never jumping to this location.

Another limitation is the Loader bootstrap code relocates itself to address 4000, so that the top of memory will be free for use by the loader, since typically most 6x09 roms load at the top of memory. Therefore any contiguous load that copies over the first 60 bytes or so starting from 4000 will also cause a crash to happen.

Lastly, the IO range starts at A000, and there is no way to tell the loader to skip a segment of memory, therefore the 64K physical top addresses starting at B000 are realistically the only addresses that make sense, as the loader has to load contiguously until it reaches the reset vector.

The best way to avoid these limits is to load the system to the top of the RAM, finishing at FFFF, doing so will guarantee that FFFE an FFFF are updated with something (the address of the start of your system code should be stored there as it is the reset vector).

If further areas of ram need to be loaded during staging the best way to do this, is by loading enough rom space with the loader to interface with the iOS LBA interface, and loading the remaining data from the first LBA device image in the directory using a custom loader.

Further notes on iOS and how it interacts with system variables:

Once a ROM image is loaded with loader, the directory the rom is contained in becomes the location for that ROM's LBA images. In other micro controller based designs like the Z80-MBC2, which use iOS, different image naming was used to separate different environments, as all of the images were stored in the root of the SD card. With the HB63C09M, all of the system images follow the same naming convention for every instance: DS0NXX.DSK, where XX is any number between 0 and 99, however the image for the specific rom being loaded will be picked from the same directory the rom is loading from. For example if the directory name is "TEST" all of the LBA images for the ROM stored in "TEST will be stored at /TEST/<images here>, this means the full path of the first LBA image for the rom stored in TEST will be /TEST/DS0N00.DSK.

The LBA (logical block addressed) images must all be laid out as follows:

- 512 tracks (0-511), selected by a two byte write to SELTRACK in big endian format (see table a1)
- 32 sectors per track (0..31), selected by a 1 byte write to SELSECT (see table a1)
- 512 bytes per sector, which must be read into ram or written to sd consecutively, and completely (see 012129)

Therefore each LBA file must be 8,388,608 bytes exactly (regardless of any logical layout within your operating system).

Appendix A: Arduino Sketch notes and overview of functions

sections marked with an * are part of iOS by Fabio DeFabis, please see attribution folder in repository. The Base address of the controller is 0xA000, each function below is listed as a single byte address based off this base address. For example TX SERIAL is 0xA001 but listed at 0x01.

WRITE OPERATIONS:

0x00: UART CONTROL {RESERVED}

FUNCTION:
This allows for enabling interrupts for interrupt controlled serial operations, exactly in the same manor as the 6850. Please refer to the 6850 data sheet for full details The only bits this currently responds to is 5,6 and 7.

In general if bit 5 and 6 are 1 and 0 respectively the uart will enable tx interrupts. Additionally, if bit 7 is 1 rx interrupts will be enabled.

SOURCE COMMENTS:

```
                // store the control req register bits for use by the wrapper
```

0x01: TX SERIAL

FUNCTION:

Sends a byte to the TX buffer internal to the Atmega32

SOURCE COMMENTS:

```
case 0x01:
  // send a byte to the terminal.
  // TX SERIAL
```

0x02: SELDISK*

FUNCTION:

Selects the LBA image in the same directory as the system rom used to boot. This value should be 0-99. any value not within this range will return an error.

SOURCE COMMENTS:
```
case 0x02:
// DISK EMULATION
// SELDISK - select the emulated disk number (binary). 100 disks are supported [0..99]:
//
//                I/O DATA:    D7 D6 D5 D4 D3 D2 D1 D0
//                            ------------------------------------------------------
//                             D7 D6 D5 D4 D3 D2 D1 D0    DISK number (binary) [0..99]
//
//
// Opens the "disk file" correspondig to the selected disk number, doing some checks.
// A "disk file" is a binary file that emulates a disk using a LBA-like logical sector number.
// Every "disk file" must have a dimension of 8388608 bytes, corresponding to 16384 LBA-like logical sectors
//  (each sector is 512 bytes long), correspinding to 512 tracks of 32 sectors each (see SELTRACK and
//  SELSECT Opcodes).
// Errors are stored into "errDisk" (see ERRDISK Opcode).
//
//
```

```
//
// ...............................................................................
//
// "Disk file" filename convention:
//
// Every "disk file" must follow the sintax "DSsNnn.DSK" where
//
//     "s" is the "disk set" and must be in the [0..9] range (always one numeric ASCII character)
//     "nn" is the "disk number" and must be in the [00..99] range (always two numeric ASCII characters)
//
// ...............................................................................
//
//
// NOTE 1: The maximum disks number may be lower due the limitations of the used OS (e.g. CP/M 2.2 supports
//         a maximum of 16 disks)
// NOTE 2: Because SELDISK opens the "disk file" used for disk emulation, before using WRITESECT or READSECT
//         a SELDISK must be performed at first.
// NOTE 3: vBios uses directories to separate the emulated file systems with their boot images]'
//         each fill will be named DS0N00.DSK, DS0N01.DSK, DS0N02 ... and so on.  The file names
//         will follow this convetion for each subdirectory that uses i0S commands inside their ROMS
```

0x03 – SELTRACK*

FUNCTION:

Select the active track to be read / written within the active LBA image. Each LBA image has 512
tracks 0-511. Data is transferred to this simulated register in a two byte write operation to address
A003. The format is MSB followed by LSB (16 bit number). The firmware keeps track of which
byte is being written. No other IO controller operation should take place during this operation.
Incomplete operations will generate an error.

SOURCE COMMENTS:
```
case 0x03:
 // DISK EMULATION
 // SELTRACK - select the emulated track number (word split in 2 bytes in sequence: DATA 0 and DATA 1):
 //
 //              I/O DATA 0:  D7 D6 D5 D4 D3 D2 D1 D0
 //                           ---------------------------------------------------------
 //                           D7 D6 D5 D4 D3 D2 D1 D0    Track number (binary) MSB [0..1]
 //
 //              I/O DATA 1:  D7 D6 D5 D4 D3 D2 D1 D0
 //                           ---------------------------------------------------------
 //                           D7 D6 D5 D4 D3 D2 D1 D0    Track number (binary) LSB [0..255]
 //
 //
 // Stores the selected track number into "trackSel" for "disk file" access.
 // A "disk file" is a binary file that emulates a disk using a LBA-like logical sector number.
 // The SELTRACK and SELSECT operations convert the legacy track/sector address into a LBA-like logical
 //  sector number used to set the logical sector address inside the "disk file".
 // A control is performed on both current sector and track number for valid values.
 // Errors are stored into "diskErr" (see ERRDISK Opcode).
 //
 //
 // NOTE 1: Allowed track numbers are in the range [0..511] (512 tracks)
 // NOTE 2: Before a WRITESECT or READSECT operation at least a SELSECT or a SELTRAK operation
 //         must be performed
 // NOTE 3: Big endian numbering so MSB is written first. (this differs from the Z80MBC and the V20MBC)
```

0x04 – SELSECT*

FUNCTION:

Selects the active sector within the active track, for reading and writing.   There are 32 sectors per track (0-31). Data is transferred to this simulated register with one write to **A004.** Any value not within the range of 0 and 31 will generate an error.

SOURCE COMMENTS:

```
case 0x04:
 // DISK EMULATION
 // SELSECT - select the emulated sector number (binary):
 //
 //                 I/O DATA:  D7 D6 D5 D4 D3 D2 D1 D0
 //                            ---------------------------------------------------------
 //                            D7 D6 D5 D4 D3 D2 D1 D0    Sector number (binary) [0..31]
 //
 //
 // Stores the selected sector number into "sectSel" for "disk file" access.
 // A "disk file" is a binary file that emulates a disk using a LBA-like logical sector number.
 // The SELTRACK and SELSECT operations convert the legacy track/sector address into a LBA-like logical
 //  sector number used to set the logical sector address inside the "disk file".
 // A control is performed on both current sector and track number for valid values.
 // Errors are stored into "diskErr" (see ERRDISK Opcode).
 //
 //
 // NOTE 1: Allowed sector numbers are in the range [0..31] (32 sectors)
 // NOTE 2: Before a WRITESECT or READSECT operation at least a SELSECT or a SELTRAK operation
 //         must be performed
```

0x05 – WRITESECT*

FUNCTION:

Write a 512 byte sector to the SD card. The CPU must perform 512 consecutive writes to the address **A005.** The firmware keeps track of the current byte to write, and updates the LBA image every 32 bytes written. Each transfer must be 512 bytes long or iOS will generate an error. No other io controller operation can take place during this exchange.

SOURCE COMMENTS:

```
case 0x05:
 // DISK EMULATION
 // WRITESECT - write 512 data bytes sequentially into the current emulated disk/track/sector:
 //
 //              I/O DATA 0: D7 D6 D5 D4 D3 D2 D1 D0
 //                          ---------------------------------------------------------
 //                          D7 D6 D5 D4 D3 D2 D1 D0    First Data byte
 //
 //                   |            |
 //                   |            |
 //                   |            |                <510 Data Bytes>
 //                   |            |
 //
 //
 //              I/O DATA 511: D7 D6 D5 D4 D3 D2 D1 D0
 //                            ---------------------------------------------------------
 //                            D7 D6 D5 D4 D3 D2 D1 D0    512th Data byte (Last byte)
 //
 //
```

```
//
//
// Writes the current sector (512 bytes) of the current track/sector, one data byte each call.
// All the 512 calls must be always performed sequentially to have a WRITESECT operation correctly done.
// If an error occurs during the WRITESECT operation, all subsequent write data will be ignored and
//  the write finalization will not be done.
// If an error occurs calling any DISK EMULATION Opcode (SDMOUNT excluded) immediately before the WRITESECT
//  Opcode call, all the write data will be ignored and the WRITESECT operation will not be performed.
// Errors are stored into "diskErr" (see ERRDISK Opcode).
//
// NOTE 1: Before a WRITESECT operation at least a SELTRACK or a SELSECT must be always performed
// NOTE 2: Remember to open the right "disk file" at first using the SELDISK Opcode
// NOTE 3: The write finalization on SD "disk file" is executed only on the 512th data byte exchange, so be
//         sure that exactly 512 data bytes are exchanged.
```

0x3B – TIMRCNT

FUNCTION:

This register controls the timing interrupt generation. Simply setting bit 0 will cause the controller to pul the /IRQ line low on the 63C09 based on the value set in SYSTICK. The default configured value of SYSTICK is 10 msec. This value can be set to any number between 1 and 255 by writing to location *0xA03C.*

BIT	Function
0	Start (1) Timer / Stop Timer (0)

SOURCE COMMENTS:

```
case 0x3B:
/* TIMRCNT --
 * TIMER CONTROL REGISTER (TCR):
 * Controls the timer operation, including starting, stopping, resetting, and enabling interrupts.
 *
 *               I/O DATA:    D7 D6 D5 D4 D3 D2 D1 D0
 *                            ----------------------------------------------------
 *
 *                               ...             D1 - D7 are future use, write 0's
 *
 *                            X  X  X  X  X  X  X  0    Timer interrupts disabled
 *                            X  X  X  X  X  X  X  1    Timer interrupts enabled
 *
 *
 *
 *
 * DESCRIPTION:
 * - Bit 0: Controls interupts, (1) enables interputs or (0) disables interupts
 */
```

0x3C – SETTICK

FUNCTION:

This register sets the max value of the systick counter. When this number is reached the controller sends a low signal on the /irq line.   The counter default value is 10 msec. This value can be set to any value between 1 and 255. a setting of 0 is ignored.

SOURCE COMMENTS:

```
case 0x3C:
// Set Sys tick timer
// SETTICK - set the Systick timer time (milliseconds)
//
//                  I/O DATA:    D7 D6 D5 D4 D3 D2 D1 D0
//                              -------------------------------------------------------
//                               D7 D6 D5 D4 D3 D2 D1 D0    Systick time (binary) [1..255]
//
// Set/change the time (millisecond) used for the Systick timer.
// At reset time the default value is 10ms.
// See SETIRQ and SYSIRQ Opcodes for more info.
//
// NOTE: If the time is 0 milliseconds the set operation is ignored.
```

0x3E – LOADERR

FUNCTION:
This is the loader register, it performs some high risk, very low level operations and must be unlocked by first writing 255 (0xFF) to **A03E**. Once unlocked it can perform the following:

WRITE BYTE	OPERATION
01	Resets the 63C09 CPU at the end of the active cycle, this will not reset the IO controller (soft reset).
02	Halts the 63C09 CPU at the end of the active cycle – this state remains until a system reset occurs
FF	Unlocks the register, required for loading memory (see 0x3E – LOADER under read operations)

SOURCE COMMENTS:

```
case 0x3E:
// LOADERR --
/*
 *   This performs some low level operations (high risk!!) understand how these work before using
 *   +--------------+----------------------------------------------------------+
 *   | write byte:  |   Operation:                                             |
 *   +--------------+----------------------------------------------------------+
 *   |     01       |   Resets only the 6309, this will not reset the AVR      |
 *   +--------------+----------------------------------------------------------+
 *   |     02       |   halts the 6309, remains halted until next sys_reset    |
 *   +--------------+----------------------------------------------------------+
 *   |              |   Unlocks the Loader enviornment for the above, or to    |
 *   |     FF       |   stage the computer. (see read operation 0x03E)         |
 *   +--------------+----------------------------------------------------------+
 *
 */
```

READ OPERATIONS CONTINUED ON NEXT PAGE

READ OPERATIONS:

0x00 UARTSTAT

FUNCTION:

This is the 6850 Wrapper status register, it behaves mostly like a 6850 would for bit's 0 and 1. Since the atmega32 has a FIFO buffer which is configured internally. The Status register is read by reading **A000,** it has the following differences:

Bit 0: Receive data register full (this is set if there is data waiting in the RX buffer)

Bit 1: transmit data empty (this is set if TX buffer has space).

Bits 2 through 3 are depreciated analog modem operations

Bit 5: buffer over/under run (not implemented)

Bit 6: depreciated analog modem operation

Bit 7: irq - indicates if an IRQ is sent from the UART wrapper

SOURCE COMMENTS:

```
case 0x00:
 // UARTSTAT
 //This is the UART status register it is simalar to the 6850 (however has a built in buffer TX & RX)
 // bit 0 = Receive Data register full (this is set if there is data waiting)
 // bit 1 = transmit data empty (this is set if TX buffer has space)
 // bit 2 = depriciated
 // bit 3 = depriciated
 // bit 4 = depricaited
 // bit 5 = buffer over/underrun - not yet implimented
 // bit 6 = depricated
 // bit 7 = irq

 // buffers are manually set to 64 bytes, testing has shown that there is little chance of an underun / overflow
 // set at this level. /DCD, /CTS and FE are depreciated.

 // TODO - write code for bits 5
```

0x02 – ERRDISK*

FUNCTION:

ERRDISK outputs the last iOS operation's disk error (or if there was no error by returning 0). These operations include SELDISK, SELSECT, SELTRACK, WRITESECT, READSECT. Due to the way that SDMOUNT works, it must not be used to read the error code for SDMOUNT, which is itself a read operation, and returns it's own error code. Read this register by reading **A002** These codes are 0 – 19:

00 – NO ERROR
01 – DISK ERROR
02 – NOT READY
03 – NO FILE
04 – NOT OPENED
05 – NOT ENABLED
06 – NO FILE SYSTEM
16 – ILLEGAL DISK NUMBER
17 – ILLEGAL TRACK NUMBER
18 – ILLEGAL SECTOR NUMBER
19 – UNEXPECTED EOF

SOURCE COMMENTS:

```
case 0x02:
 // DISK EMULATION
 // ERRDISK - read the error code after a SELDISK, SELSECT, SELTRACK, WRITESECT, READSECT
 //           or SDMOUNT operation
 //
 //              I/O DATA:    D7 D6 D5 D4 D3 D2 D1 D0
 //                          --------------------------------------------------------
 //                           D7 D6 D5 D4 D3 D2 D1 D0    DISK error code (binary)
 //
 //
 // Error codes table:
 //
 //    error code   | description
 // ------------------------------------------------------------------------------------------------
 //        0        | No error
 //        1        | DISK_ERR: the function failed due to a hard error in the disk function,
 //                 |  a wrong FAT structure or an internal error
 //        2        | NOT_READY: the storage device could not be initialized due to a hard error or
 //                 |  no medium
 //        3        | NO_FILE: could not find the file
 //        4        | NOT_OPENED: the file has not been opened
 //        5        | NOT_ENABLED: the volume has not been mounted
 //        6        | NO_FILESYSTEM: there is no valid FAT partition on the drive
 //        16       | Illegal disk number
 //        17       | Illegal track number
 //        18       | Illegal sector number
 //        19       | Reached an unexpected EOF
 //
 //
 //
 //
 // NOTE 1: ERRDISK code is referred to the previous SELDISK, SELSECT, SELTRACK, WRITESECT or READSECT
 //         operation
 // NOTE 2: Error codes from 0 to 6 come from the PetitFS library implementation
 // NOTE 3: ERRDISK must not be used to read the resulting error code after a SDMOUNT operation
 //         (see the SDMOUNT Opcode)
```

0x03 – READSECT*

FUNCTION:

Read a 512 byte sector from the SD card. The CPU must perform 512 consecutive reads to the address **A003.** The firmware keeps track of the current sector byte to read, and reads 32 byte's of the sector from the card into the AVR memory at a time. Each transfer must be 512 bytes long or iOS will generate an error. No other IO controller operation can take place during this exchange, or the controller will produce an error.

SOURCE COMMENTS:

```
case 0x03:
 // DISK EMULATION
 // READSECT - read 512 data bytes sequentially from the current emulated disk/track/sector:
 //
 //                  I/O DATA:    D7 D6 D5 D4 D3 D2 D1 D0
 //                             -------------------------------------------------------
 //                  I/O DATA 0  D7 D6 D5 D4 D3 D2 D1 D0    First Data byte
 //
 //                             |               |
 //                             |               |
 //                             |               |                  <510 Data Bytes>
 //                             |               |
 //
 //                  I/O DATA 127  D7 D6 D5 D4 D3 D2 D1 D0
 //                             -------------------------------------------------------
 //                                D7 D6 D5 D4 D3 D2 D1 D0    512th Data byte (Last byte)
 //
 //
 // Reads the current sector (512 bytes) of the current track/sector, one data byte each call.
 // All the 512 calls must be always performed sequentially to have a READSECT operation correctly done.
 // If an error occurs during the READSECT operation, all subsequent read data will be = 0.
 // If an error occurs calling any DISK EMULATION Opcode (SDMOUNT excluded) immediately before the READSECT
 //  Opcode call, all the read data will be will be = 0 and the READSECT operation will not be performed.
 // Errors are stored into "diskErr" (see ERRDISK Opcode).
 //
 // NOTE 1: Before a READSECT operation at least a SELTRACK or a SELSECT must be always performed
 // NOTE 2: Remember to open the right "disk file" at first using the SELDISK Opcode

 if (!ioByteCnt)
```

0x04 – SDMOUNT*

FUNCTION:

Mounts an SD Volume, normally only used by the firmware, iLoad is not implemented in vBIOS.

SOURCE NOTES:

```
case 0x04:
 // DISK EMULATION
 // SDMOUNT - mount a volume on SD, returning an error code (binary):
 //
 //                  I/O DATA 0: D7 D6 D5 D4 D3 D2 D1 D0
 //                             -------------------------------------------------------
 //                                D7 D6 D5 D4 D3 D2 D1 D0    error code (binary)
 //
 //
 //
 // NOTE 1: This Opcode is "normally" not used. Only needed if using a virtual disk from a custom program
 //         loaded with iLoad or with the Autoboot mode (e.g. ViDiT). Can be used to handle SD hot-swapping
 // NOTE 2: For error codes explanation see ERRDISK Opcode
 // NOTE 3: Only for this disk Opcode, the resulting error is read as a data byte without using the
 //         ERRDISK Opcode
```

0x3B – TIMRSTA

FUNCTION:

This is the timer status register. A value on bit 0 indicates an interrupt has been triggered. Simply reading the timer status register clears the timing interrupt. The interrupt handler within the controller manages holding the line low, and will not release /irq unless the remaining interrupts are also clear. The ISR should manage all set interrupts, and not just timing.

BIT	
0	Timer is started (1) / Timer is stopped (0)

SOURCE NOTES:

```
case 0x3C:
// Set Sys tick timer
// SETTICK - set the Systick timer time (milliseconds)
//
//                   I/O DATA:    D7 D6 D5 D4 D3 D2 D1 D0
//                                -------------------------------------------------------
//                                D7 D6 D5 D4 D3 D2 D1 D0    Systick time (binary) [1..255]
//
// Set/change the time (millisecond) used for the Systick timer.
// At reset time the default value is 10ms.
// See SETIRQ and SYSIRQ Opcodes for more info.
//
// NOTE: If the time is 0 milliseconds the set operation is ignored.
```

0x3C – SAMPTICK

FUNCTION:

This register contains the current running value of SYSTICK in ms. This can be useful when an arbitrary number is needed for a routine, ie psudorandom / retry count. This value is going to be in ms and is any number between 0 and SYSTICK

SOURCE NOTES:
```
case 0x3C:
// SAMPTICK
// sample the systick timer, may be useful for retry counter, psudonumber generation etc...
// will be a number from 0 - sTickTime.
```

0x3E – LOADER

FUNCTION:

The loader takes the system bootstrap rom from the SD card, and loads it to memory. In order for the loader to work the computer has to be in bootstrap mode, with the system rom selected by the SD Card Library. This occurs right the controller is reset, and it is a mode the system can not currently enter post staging. In addition to this, the low level system control register LOADERR must be unlocked to allow staging to complete. This is so it can not accidentally be unlocked which could lead to a non-recoverable state, requiring a hard reset.

The process starts by checking for a writable area at **FFFF.** The controller writes the number 42 to this location and checks to see if it can read 42. Once it is determined that the ram area is present at the top of memory the controller outputs a message via serial to the user indicating that the staging has begun.

To stage the computer, the controller assumes control of the address bus, and strobes a very short 46 byte program into memory at **FFC0:**

```
;blockcopy.asm

Loader: equ $4000                      ;copy loader to this location
loadsm: equ $84                              ;code to modify lda,x+ → lda,x
loadfm: equ $A03E                      ;address of the byte loader

        org $FFC0                      ;MCU Loads blockcopy here.
        ldx #CopyBlock                      ;copy code start address to x
        ldy #loader                    ;destination address into y
        ldw  #((lstByt+2)-Copyblock)   ;number of bytes to copy → W

;; we need to relocate the loader to the static block so we can load
;; rom space.


CopyBlock:

        lda  ,x+                              ;transfer a byte
        sta  ,y+                              ;store a byte
        decw                           ;keep track of where we are here AND
                                             ;inside the MCU.

LstByt: bne CopyBlock                        ;if were not done keep going


;; This next is a bit tricky, the byte loader sends a new byte at each read
;; attempt so when we are reading from the address stored in x multiple
;; times each time this is a new value.  See the Arduino sketch for the
;; specific structure of the stored data (TLDR - byte 1 and 2 = destination
;; address,  byte 3 and 4 contain the number of bytes - and the following
;; data is the actual ROM code.) The read attempt must be continuous and
;; consecutive or the loader will reset.

;; Keep in mind - the destination address is in y, the byte loader address
;; is in x, and the byte count is stored in w.  When the code jumps to the
;; loader this is a modified version of the code above this comment block
;; from CopyBlock to LstByt almost exactly -- with the exception that
```

```
;; lda ,x+ has now become lda ,x by virtue of the last two lines before the
;; jump.

        ldx   #loadfm                            ; set the start address to the loader
        lda   #255                     ; unlock code, magic number to loader
        sta   ,x                                 ; ..
        lda   ,x                                 ; load destination address into D
        ldb   ,x                                 ; ..
        tfr   d,y                                ; we need the a register
        lde   ,x                                 ; byte count into w
        ldf   ,x                                 ; ..
        lda   #loadsm                            ; load self modifying code to a
        sta   loader+1                           ; modify the code lda,+x → lda,x
        jmp   loader                   ; jump to loader and start copying

;;  from this point CPU is in free run, but after 1/4 E the AVR should reset
;;  the system as it has counted along with the w register. From this point
;;  the 63C09 should bootstrap the system by loading 1 byte at a time to the
;;  location set by the y register typically to the top of memory.
```

After loading the blockcopy routine into memory, the controller sets the address bus to **FFFE** (and **FFFF**) to set the reset vector to **FFCO.**

It checks to see if the SD card can be mounted, if it can not find the SD card it will ask the user to check to see if the SD card is installed via the serial terminal. Once it can read from the SD card the computer will proceed to checking the status of the EEPROM space.

The EEPROM inside the controller stores the values used by the bootstrap. It runs a checksum check on the data at the beginning of this space – if it is found to be non-valid it will dump the user into the vBIOS mode which will allow the user to enter these values to the EEPROM.

Default values are:

        START: C000
        SIZE: 4000
        FILE: BIOS.BIN
        PATH: /

BIOS.BIN must be saved on the SD card, in the root of the SD card for these values to work. This can be found at: **<location of repository>HB63C09/System Roms/BIOS.BIN**

using vBIOS commands SET, COMMIT and QUIT you can save the values to the correct settings and commit them. Every time you commit values to the EEPROM, this updates the checksum information within the EEPROM.

Once the EEPROM data is loaded into the controllers system bootstrap variables, the volume bootstrap file is loaded off the SD card and then it's index is reset to zero. If this works, the system controller gives up the bus, and lets the CPU out of reset. The 63C09 then jumps to the reset vector and loads the blockcopy routine that the controller had previously stated to memory.

Appendix B PCB header descriptions:

The following descriptions are based on the board being positioned with the side headers for the serial busses (the SD card and the TTL to USB header) on the left hand side. With the user sitting in front of the side of the board with the reset button.

JMP 1:

A two pin jumper block, closing this jumper powers the computer from the TTL to USB device.
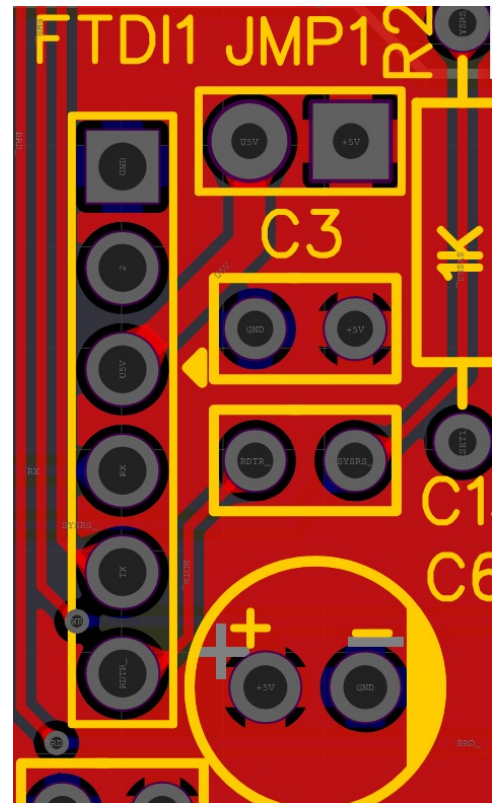
FTDI :

matches the pin out of the FT232RL Mini USB to TTL Serial Converter Adapter.

(From the top)

pin 1 – GND
pin 2 – CTS (not connected)
pin 3 – 5V (Activated through JMP 1)
pin 4 – RX (Connects to TX on USB → TTL)
pin 5 – TX (Connects to RX on USB → TTL)
pin 6 – DTR/RTS (Connects to DTR/RTS)

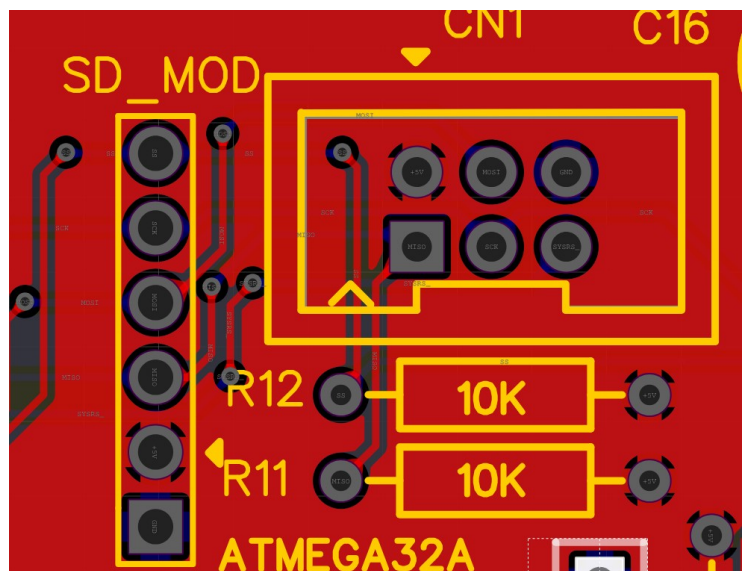*+5v denoted by an arrow point on PCB.*

SD Card module:

A standard 5v SD card module with a level shifter

from the top

pin 1 – SS
pin 2 – SCK
pin 3 – MOSI
pin 4 – MISO
pin 5 – 5V
pin 6 – GND

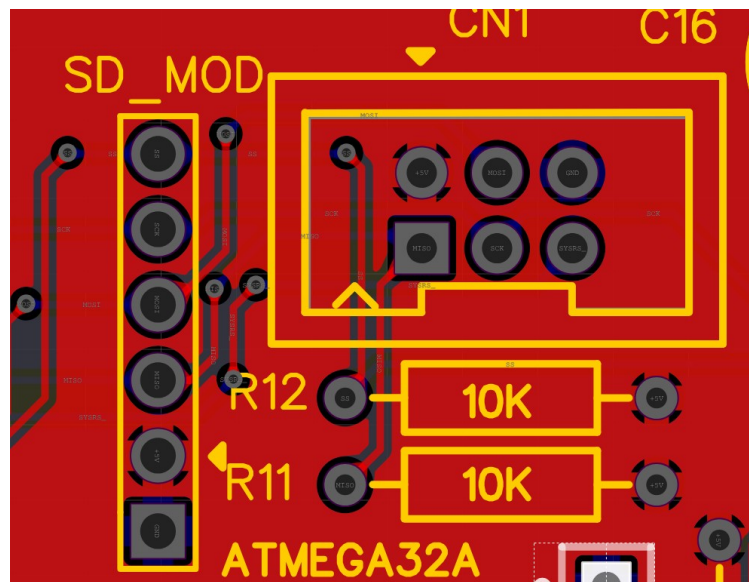*+5v is denoted on pcb with an arrow point*

ICSP Connection

from left to right:

top row:+5v, MOSI, GND
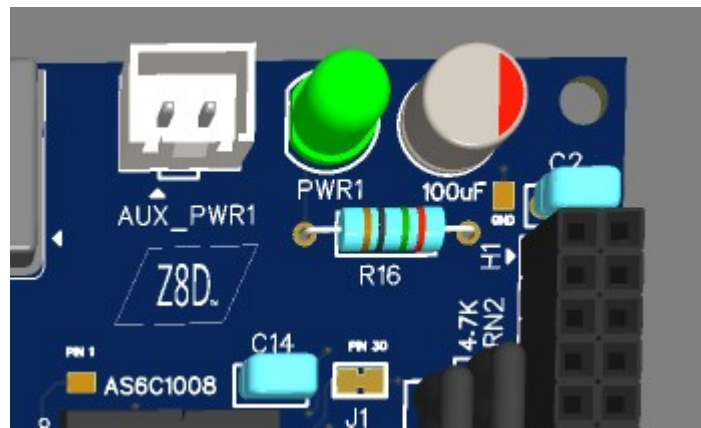bot.row:MISO,SCK, RESET (system)

+5V is denoted by an arrow point on the
PCB

AUX Power Header:

This is a two pin aux power JST connector.
This is for powering the board remotely when
powering from USB is not feasible.

+5V denoted by an arrow point on the pcb

In addition to the headers there are several other solder-able pads, on the board which break out
signals for banking, the system (controller) reset and the SD LED.   On the 4.2 revision of the board
there is a trace jumper for disconnecting pin 30 of the SRAM chip from +5V This is for allowing
for the possibility of a 1MB expansion.

31

System bus connector

The system bus connector is a 40 pin dual inline female IDC connector. Not to be confused with the raspberry pi, IDE or the expansion header for the HB6809 (these are not the same connector pin out). Connecting them in this manor could cause damage to either interface.

PIN	LEFT SIDE	RIGHT SIDE
1	5V	GND
2	/HLT	S.CLK
3	/RES	E
4	/DMA	/IRQ
5	A0	A1
6	A2	A3
7	A4	A5
8	A6	A7
9	A8	A9
10	A10	A11
11	A12	A13
12	A14	A15
13	D0	R//W
14	D1	BRD_
15	D2	BWR_
16	D3	XSIN_
17	D4	BS
18	D5	BA
19	D6	SCL
20	D7	SDA
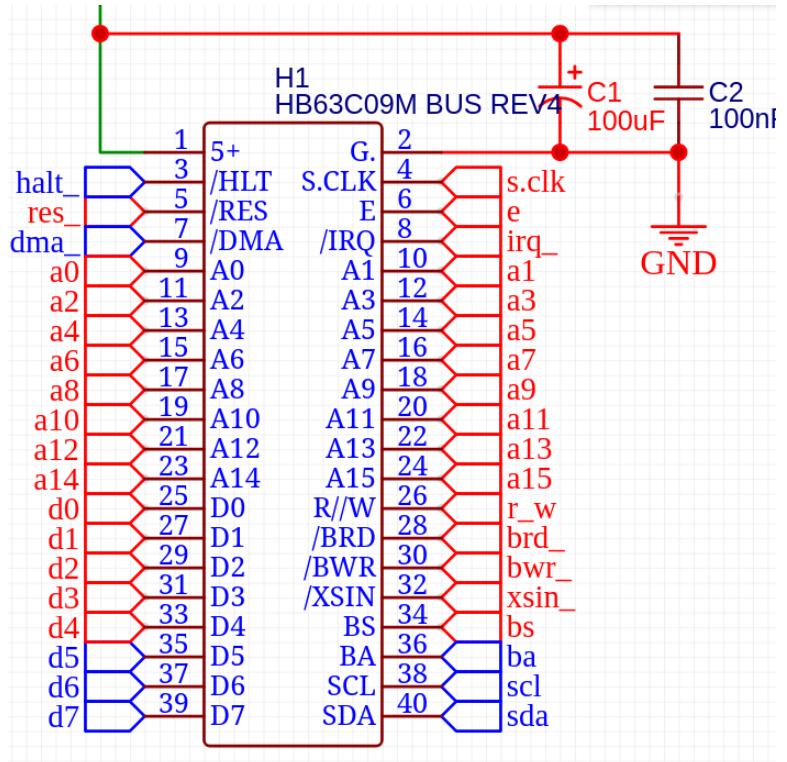
Inputs on the bus connector **requiring an "open collector"** output:

/halt, /res, /dma, /irq, /XSIN

CAUTION!! Driving any one of these pins high may cause a direct short to ground through another chip. Please do not try to connect them to "push pull" outputs (even for testing) this may permanently damage inputs on the CPU, MCU or the Inverter package. Additionally – assure any output connected to the connector is in tri-state while /reset is LOW (including reset). Failure to do this can cause the computer to not boot.

See architecture description for more information on using the connector.

Appendix C Interrupt Errata / mS Interval timer:

The controller can generate a low state on the /IRQ line by pulling the line low. It does this currently for 3 different interrupts the 2 68B50 RX and TX interrupts, and the mS interval timer. When an interrupt happens an internal interrupt register bit is set. To the user the there are 2 status registers to watch:

- The UART status register at 0xA000
- The timer status register at 0xA03B

UART:

The reference the 6850 data sheet for more information, however currently just the TX and RX interrupts are supported. The best way to view the UART when using interrupt controlled serial is to consider the next byte on the buffer to be the TX and RX register in the uart chip. The internal UART inside the controller will continue to buffer and hold characters as they are sent up to the limit of 64kb. Currently there is not a way to handle overflow however run fast enough most modern PC emulators will absolutely clear the buffer faster than the HB63C09M can process the characters.

To clear interrupts from the TX register, sending a byte clears the interrupt bit, but the status register isn't actually updated until the top of the next fast poling cycle. Conversely the RX interrupt is cleared when the uart status register is read. Similarly the status register is updated at the top of the next fast poling cycle. While the interrupt flag is cleared immediately, which may clear the IRQ line from the CPU. See the section on interrupt handling, to understand how the IRQ generation is decided.

TIMER:

The controller can simulate a millisecond timing interrupt using a programmable interrupt timer. The default timer is set to **10 ms**. You can enable timing interrupts by setting **bit 0** of **0xA03B**. Once set, the controller will generate an interrupt based on the value stored in **SYSTICK**. By default, **SYSTICK** is set to **10 ms**, but you can change this by writing a value to **0xA03C**. This value must be between **1 and 255**, setting the interrupt interval in milliseconds.

Each time an interrupt is triggered, the controller sets **bit 0** in the **Timing Status Register**, which can be checked by reading **0xA03B**. Once read, the controller clears this flag and resets the internal interrupt handling. If no other interrupts are pending, it will also raise the **/IRQ** line. However, the timer will automatically restart as soon as the interrupt is generated (i.e., this is a **count-to-limit and reset interrupt**).

The **Interrupt Service Routine (ISR)** should be kept short to prevent missing the next timer event. If the ISR performs excessive computations, delays, or lengthy I/O operations, the next timing interrupt may occur before the previous one has finished, potentially leading to missed interrupts or system instability. If precise timing is required, consider offloading complex operations to a separate process or deferring them until after the ISR completes.

INTERRUPT HANDLING:

The controller references an internal **interrupt register** that keeps track of active interrupts. This register is checked during each **fast polling cycle** to determine if the **IRQ line** should be cleared, as well as in the following cases:

- **After a character is sent**, if TX interrupts are enabled, the **TX interrupt is cleared**.
- **After the UART Wrapper status register (0xA000) is checked**, if a character was received, the **RX interrupt is cleared**.
- **After the status register is checked**, if the **timer interrupt** is cleared.

Performing a **hard reset** using the **PCB reset button** will also clear all interrupts and **re-stage** the computer.

When designing an **Interrupt Service Routine (ISR)**, keep in mind that if multiple interrupt sources are enabled, **all active interrupts must be cleared before returning to the main execution**. This ensures that the **IRQ line** is released to its normal operating state (held **high** by the pull-up resistor).

Additionally, the **IRQ line** is an **open-collector input** on the CPU and can only be **pulled low. Do not drive this line high** using a push-pull output, as this could create a **short circuit** with the microcontroller. (See notes in **Appendix B** for additional details.)