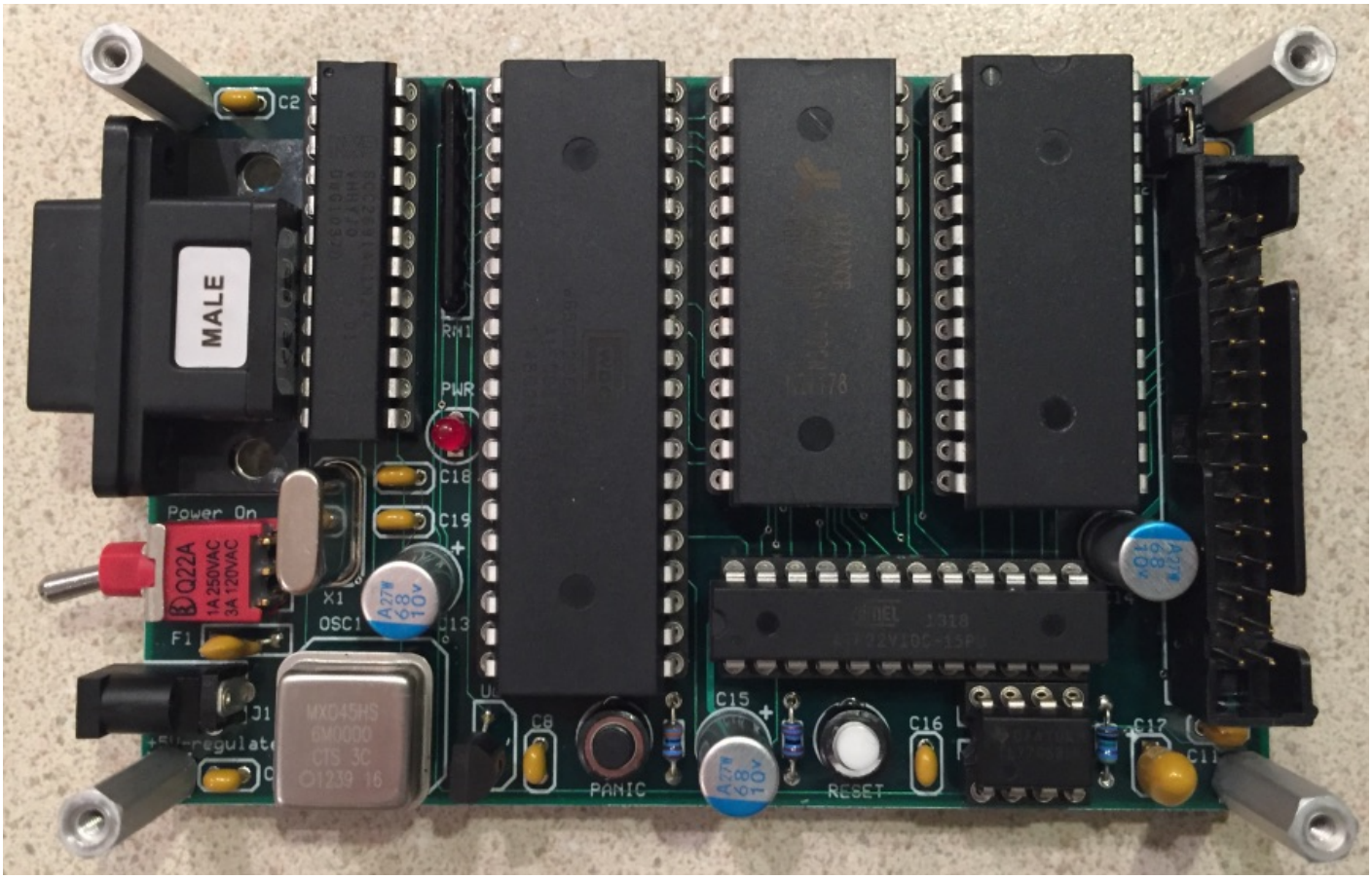
The C02 Pocket-SBC

A Pocket-sized 65C02 based computer



Basic Description:

The design goals are based on proving some changes to what I would call a “Classic 6502 System” from a hobbyist view. Specific Design Features are outlined below:

1- *A simple and flexible hardware design by utilizing a single PLD Glue chip*
2- *Using a NXP SCC2691 UART which is a departure to the 6551*
3- *A new Reset chip that provides active high and low signals (UART pre-req)*
4- *A NMI Panic switch for debugging capability – saves CPU state, restores I/O*
5- *A Bus expansion connector for additional I/O devices and capabilities*

Having discovered a hardware bug in the recently released W65C51 ACIA, it was time to move to a new UART for use as a Console for interacting with the system. BDD (from 6502.org fame) had been telling me for quite a while to move to a NXP UART. This new board design uses a NXP SCC2691. It also incorporates a FTDI USB-to-serial interface that is in a DB9 connector housing. I have been using these interface devices for a while and it made sense to include this as part of the design for the new SBC. This allows direct attachment to a USB port on any modern PC to be used as a Console. After trying several PC Terminal programs, I've settled on using ExtraPutty.

The NXP UART adds the requirement for an active high Reset signal in addition to the active low Reset signal that 65XX chips use. After some research, I settled on the Texas Instruments TL7705B. It provides a flexible delay time for holding the Reset lines active via a RC network and has open collector outputs for both Reset signals. It also supports a manual reset trigger. I opted for a Maxim DS1813 Reset chip to de-bounce the NMI line for the Panic button. The Reset timing of the TL7705B is set to be longer than the DS1813, which is fixed at around 350ms. This is done to ensure the NMI signal is not active after the Reset signal ends on power up.

To maintain a flexible design, I've implemented a single Glue chip using Atmel's ATF22V10CQZ PLD. This chip provides the qualified Read and Write signals for memory access, the UART and any other I/O devices that are Intel compatible. It also provides the chip selects for the RAM and EEPROM as well as the UART, plus four additional I/O selects available via the Bus expansion connector. All I/O selects are 32-bytes wide, which allows for a wide range of I/O devices.

The memory used for the design is an Alliance AS6C62256 32KBx8 Static RAM and an Atmel AT28H256 32KBx8 EEPROM. These are very common chips, readily available and the EEPROM can be programmed insitu with some simple programming. The remaining bits of the design includes a half-size 6MHz Oscillator to drive the CPU and PLD clock lines, a 3.6864MHz Crystal for the SCC2691 UART, a 750ma Poly-fuse to protect the board / components and a fairly large number of bypass capacitors to ensure quiet and stable board operation.

The schematic for the Pocket-SBC was done using ExpressPCB and the 4-layer PCB layout was also done with ExpressPCB. I opted to use their MiniBoard Pro service which provides 3 high-quality 4-layer boards with solder mask and a single top side silk screen. A tight layout using all DIP parts just fits on the board which is limited to 350 holes and a size of 2.5" x 3.8". The final PCB layout came in at 349 holes!

I spec'd high quality parts for the board and used high quality sockets for all DIP parts as well. The board has proven to be stable and reliable running at 6MHz. This limit is imposed by the NXP SCC2691 UART, as it is an older UART from Philips. While 6MHz may not seem very quick, note that most early 6502 computers ran at 1MHz. Even the famous Atari vector-based Tempest Arcade Game used a 6502 running at 1.8MHz. So, this small board can do quite a bit standalone and adding other hardware devices and supporting software will extend that capability.

Hardware Configuration:

The Hardware Configuration is defined by the logic programmed into the PLD Glue chip. Back in the latter half of the 1980's, I made a simple design for the Rockwell R65C02. This used a few 74xx Logic chips for Glue logic and worked well. I did a new implementation of this about four years ago using WDC's W65C02 and 74HCxx logic chips. Based on this, I have my own (logical) view on how to arrange the memory / hardware map for the 65C02, which is limited to 64KB total addressing which includes all memory and hardware devices.

Some specific features of the 65C02 dictate that memory exists at the beginning of the memory address space, like Page Zero and the Processor Stack on Page One. It also dictates that some form of ROM exists at the end of memory address as the hardware vectors for Reset, NMI and IRQ are the last 6 addresses of the 64KB address space. To keep this simple, I split the address range between RAM and ROM, 32KB each. As the first 512 bytes need to be RAM for basic system operation, the upper 512 bytes was mapped for ROM and I/O devices as follows:

1- *$0000 - $7FFF = RAM*
2- *$8000 – $FDFF = ROM*
3- *$FE00 – $FE9F = I/O*
    a. *I/O-0 = $FE00-$FE1F (expansion bus)*
    b. *I/O-1 = $FE20-$FE3F (expansion bus)*
    c. *I/O-2 = $FE40-$FE5F (expansion bus)*
    d. *I/O-3 = $FE60-$FE7F (expansion bus)*
    e. *I/O-4 = $FE80-$FE9F (SCC2691 UART)*
4- *$FEA0 - $FFFF = ROM*

Using the above as a reference, I tend to grow System RAM usage from the bottom up and ROM usage from the top down. This provides a larger contiguous memory space which allows for larger user programs. In the initial release, the BIOS uses the first 1024 bytes for Page Zero, Processor Stack, Console FIFO buffers and Vector, Configuration and Buffers for Monitor routines. I've also reserved the second 1024 bytes for future expansion. This provides for user RAM starting at $0800 and extending to $7FFF, or 30KB. The ROM side is organized similarly. The top 2KB is for BIOS and I/O, followed by 6KB for the base Monitor. This leaves 24KB for additional ROM based software.

The Pocket-SBC requires a single regulated power supply of 5.0 Volts DC and should be a minimum of 1.0 ampere of available current. Note that the USB Console port does not provide power to the board. At idle operation, the Pocket-SBC draws approximately 45ma of current, most of this drawn by the PLD chip. Also note that the FTDI USB to UART interface gets power from the USB connector on the PC, not from the SBC power supply.

Software Configuration:

The Initial Software release for the Pocket-SBC consists of two pieces of code:

1- *The C02 BIOS which resides in ROM. Reserved range from $F800 - $FFFF (2KB)*
2- *The C02 Monitor which resides in ROM. Reserved range from $E000 - $F7FF (6KB)*
3- *The rest of the ROM is free for additional programs from $8000 - $DFFF (24KB)*

As the only I/O device on the Pocket-SBC is the UART, the current BIOS provides full support of the UART's Transmit and Receive functions required for a User Console and supports the UART's Timer/Counter as a 10ms Jiffy Clock. The Jiffy Clock is used for a Real Time Clock (RTC) that keeps track of time since System Cold Start in the form of Seconds, Minutes, Hours and Days (up to 65535 days) and also provides a core delay routine of 10ms that is used via the BIOS and Monitor for accurate delays with a wide range just over 491 days!

The BIOS provides an Interrupt Service Routine that supports full-duplex Transmit and Receive of the UART Console data, the Timer/Counter and Received Break via the Terminal program on the PC. The latter (Received Break) is used to break the SBC out of any running software loop.

The BIOS also uses a Vector based set of routines addresses which are held in RAM starting at location $0300. This provides entry and exit routines for the UART which can be extended for other devices and/or software functions. The default UART configuration data is held in RAM starting at location $0320. This allows the user to temporarily change the UART configuration and just call the Initialization routine to change the operating parameters. The default Console setup is for 38.4K baud rate, 8- Data bits, No Parity and 1- Stop bit. This allows for binary transfer of data between the host PC and the Pocket-SBC.

The BIOS functions are accessible via a Jump Table starting at location $FF00. There are 32 available functions calls in the Jump Table. The current BIOS version of 2.0 only uses 13 of these table entries for supporting the System startup, UART operations and entering the Monitor via the soft Vectors. The rest of the Jump Table entries are available for future expansion of the BIOS. There's still over 1KB of ROM space left is the defined BIOS memory mapping.

The Monitor provides a rich set of functions that include a set of Memory Operations, Register Operations, Timer/Delay Operations, A Macro Facility for loop runs/testing and a set of Control-key functions. It also has a table-driven Disassembler that supports the full set of WDC Opcodes and Addressing modes for WDC's W65C02S processor. The current Monitor version is 2.0 and has its core functions accessible via a Jump Table starting at location $E000. There are 32 Jump Table entries, of which the Monitor currently uses 23. The next version of the Monitor will use more of these for additional functions.

BIOS Jump Table Entries:

$FF00	Reserved for Future Expansion
$FF03	Reserved for Future Expansion
$FF06	Reserved for Future Expansion
$FF09	Reserved for Future Expansion
$FF0C	Reserved for Future Expansion
$FF0F	Reserved for Future Expansion
$FF12	Reserved for Future Expansion
$FF15	Reserved for Future Expansion
$FF18	Reserved for Future Expansion
$FF1B	Reserved for Future Expansion
$FF1E	Reserved for Future Expansion
$FF21	Reserved for Future Expansion
$FF24	Reserved for Future Expansion
$FF27	Reserved for Future Expansion
$FF2A	Reserved for Future Expansion
$FF2D	Reserved for Future Expansion
$FF30	Reserved for Future Expansion
$FF33	Reserved for Future Expansion
$FF36	Character Input (no waiting): Carry Flag set if Character is in A Register, else Clear
$FF39	Character Input: Waits for a Character and returns in A Register, Carry Flag Set
$FF3C	Character Output: Sends Character in A Register to Console, A Register preserved
$FF3F	Set Delay Time: Input for Millisecond Count and 16-bit Multiplier value (in Hex)
$FF42	Execute Millisecond Delay: 1-256 (times) 10ms (Jiffy Clock resolution)
$FF45	Execute Long Delay: 16-bit Multiplier (times) Millisecond Delay above
$FF48	Execute Extra Long Delay: 8-Bit Multiplier (times) Execute Long Delay values (in Hex)
$FF4B	Initialize Vectors: Initialize Software Vectors from ROM to RAM
$FF4E	Initialize Configuration: Initialize I/O Configuration Data from ROM to RAM
$FF51	Initialize UART: Sets up operating parameters for SCC2691
$FF54	Reset UART: Resets the SCC2691 – called before the above routine at startup
$FF57	Monitor Warm Start: Jumps to Monitor Warm Start Vector
$FF5A	Monitor Cold Start: Jumps to Monitor Cold Start Vector
$FF5D	System Cold Start: Jumps to Cold Start Routine (same as system Reset)

Note that in most cases, register contents are preserved on exit, i.e., delay routines can be called without saving register contents. They will be the same on exit as on entry. Initialization routines change the register contents as they are typically used for startup only.

Monitor Jump Table Entries:

$E000	Monitor Warm Start: Warm Start entry point – Vector points here
$E003	Monitor Cold Start: Cold Start entry point – Vector points here
$E006	Reserved for Future Expansion
$E009	Reserved for Future Expansion
$E00C	Reserved for Future Expansion
$E00F	Reserved for Future Expansion
$E012	Reserved for Future Expansion
$E015	Reserved for Future Expansion
$E018	Reserved for Future Expansion
$E01B	Reserved for Future Expansion
$E01E	Reserved for Future Expansion
$E021	Processor Status: Displays Program Counter and Processor Registers
$E024	Disassemble Instruction: Will disassemble current instruction at INDEXL/INDEXH
$E027	Increment Index: Increments INDEXL/INDEXH by 1
$E02A	Decrement Index: Decrements INDEXL/INDEXH by 1
$E02D	Read Line from Console: Reads a Line of Hex Characters for Monitor Input
$E030	Read Character from Console: Reads a Character, converts to Upper Case
$E033	Hex Input 2: Inputs up to Two Hexadecimal Characters from Console
$E036	Hex Input 4: Inputs up to Four Hexadecimal Characters from Console
$E039	Hexadecimal to ASCII: Converts a 16-bit Hexadecimal value to Decimal ASCII String
$E03C	Binary to ASCII: Converts an 8-bit Binary value to Two Hexadecimal ASCII Characters
$E03F	ASCII to Binary: Converts up to Two Hexadecimal ASCII Characters to an 8-bit Binary
$E042	Beep: Sends a Beep Control Character to the Console
$E045	Dollar: Sends a “$” to the Console
$E048	C/R Out: Sends a Carriage Return and Linefeed character to the Console
$E04B	Space: Sends an ASCII Space character to the Console
$E04E	Print Byte: Prints a Byte Value as Two ASCII Hexadecimal Characters to Console
$E051	Print Word: Prints a Word Value as Four ASCII Hexadecimal Characters to Console
$E054	Print ASCII: Prints any valid Byte value to Console as ASCII, else sends a “.”
$E057	Prompt Print: Prints a Message based on a Message number in the A Register
$E05A	Prompt Print2: Prints a Message based on PROMPTL/PROMPTH pointer
$E05D	Continue: Prompts to Continue execution, else Pops the Stack and re-enters Monitor

In most cases, required Data as input is via the A register and Y register (high order / low order). In most cases, Data as output is via the A register and Y register (high order / low order). The Monitor Source code is documented for all functions and register usage and should be consulted as required.

Monitor Commands:

Command	Parameters	Command Description
A	New Value or [return]	Display A Register – Change to New
C	Source, Target, Length	Compares two blocks of memory
D	Address	Displays 256 Bytes of memory as Hex / ASCII
E	Address	Examine/Edit Sequential Memory Locations
F	Source, Length, Value	Fills a Block of Memory with a Hex value
G	Address	Execution from Specified Address
H	Hex Data (16 bytes max)	Searches for Hexadecimal Data up to 16 bytes
I	Address – then text	Directly enter ASCII text into Memory (ESC to quit)
M	Source, Target, Length	Moves a Block of Memory from Source to Target
P	New Value or [return}	Display Processor Status Register – Change to New
R	[no input]	Displays All CPU Registers and Program Counter
S	New Value or [return]	Displays Processor Stack Pointer – Change to New
T	ASCII Data (16 chars Max)	Searches for ASCII string up to 16 bytes
X	New Value or [return]	Display X Register – Change to New
Y	New Value or [return]	Display Y Register – Change to New
(	[no input]	Resets Input FIFO - Macro input up to 127 keystrokes
)	[no input]	Starts Macro Loop – “Send Break” from Terminal to exit
,	ms Value / 16-bit Multiplier	Millisecond Delay Count (8-bit) / 16-bit Multiplier Value
.	[no input]	Executes Millisecond Delay – 8-bit Count (times) 10ms
/	[no input]	Executes Long Delay – 16-bit Multiplier (times) above
\	Hex value ($00 - $FF)	Executes ExtraLong Delay – 8-bit value (times) above
CTRL-D	Address	Disassembler – Disassembles 22 lines, N for next
CTRL-L	Address/Offset [optional]	Xmodem-CRC Loader with S-Record Support
CTRL-P	Source, Target, Length	Program EEPROM – Source must be in RAM
CTRL-Q	[no input]	Display Terse List of All Commands
CTRL-R	Y/N Prompt	Restarts System
CTRL-T	[no input]	Displays Up Time since System Start or Reset
CTRL-V	[no input]	Displays Monitor and BIOS Version
CTRL-Z	Y/N Prompt	Zero All Memory and Restart System

The C02 Monitor is a “prompting” design. This means it will prompt the user for all required input. If a “$” is presented, the required input is Hexadecimal data (0-F). The Return key is used rather than the Space bar to accept the input field and move to the next (field) or complete the command. You can use short input as well, i.e., $100 instead of $0100. All input data is validated during input, i.e., you can’t enter incorrect field data for Monitor commands. Any input error results in an audible Beep. You can Backspace as well in the case of incorrect entry parameters. (ESC) exits most every command before execution. All destructive memory operations require additional commit from user as “cont?(y/n)”. Realize however, that with the Monitor, you can overwrite things and crash the system is you’re reckless. The I/O regions are masked from the Memory Search and Display functions however, as these can put the NXP UART into a test mode!

Reference Information and Kudos:

Some Reference links are contained here and some thanks to some of the 6502.org Forum users:

BDD: for insisting on the NXP UART, debugging, some BIOS optimizations, plus, plus....

cbscpe: for cleaning up and simplifying the WinCUPL source for the PLD logic, Danke!

Dr Jefyll: for lots of clever ideas on debugging the SCC2691 BIOS (and W65C02 bugs).

barrym95838: for a short and clever routine to convert 16-bit binary to ASCII Decimal.

GarthW (aka Boss): for endless information, feedback, input, etc.

BigEd: The group encyclopedia, finds everything ;-)

The rest of crowd that looks at my posts and such without giving me grief.

Some Links I used for this project:

http://forum.6502.org/viewforum.php?f=11

http://wilsonminesco.com/

http://wdc65xx.com/

https://www.mouser.com/Electronic-Components/

https://www.nxp.com/products/analog/interfaces/uarts:MC_50931#/&page=1

http://www.ftdichip.com/Products/Modules/USBRSxxx.htm#DB9-USB

https://www.expresspcb.com/free-cad-software/

http://www.atmel.com/tools/wincupl.aspx

All the Best, Floobydust (aka KM)