COSMAC "ELF"
Microcomputer Trainer User's Manual

by Paul Schmidt

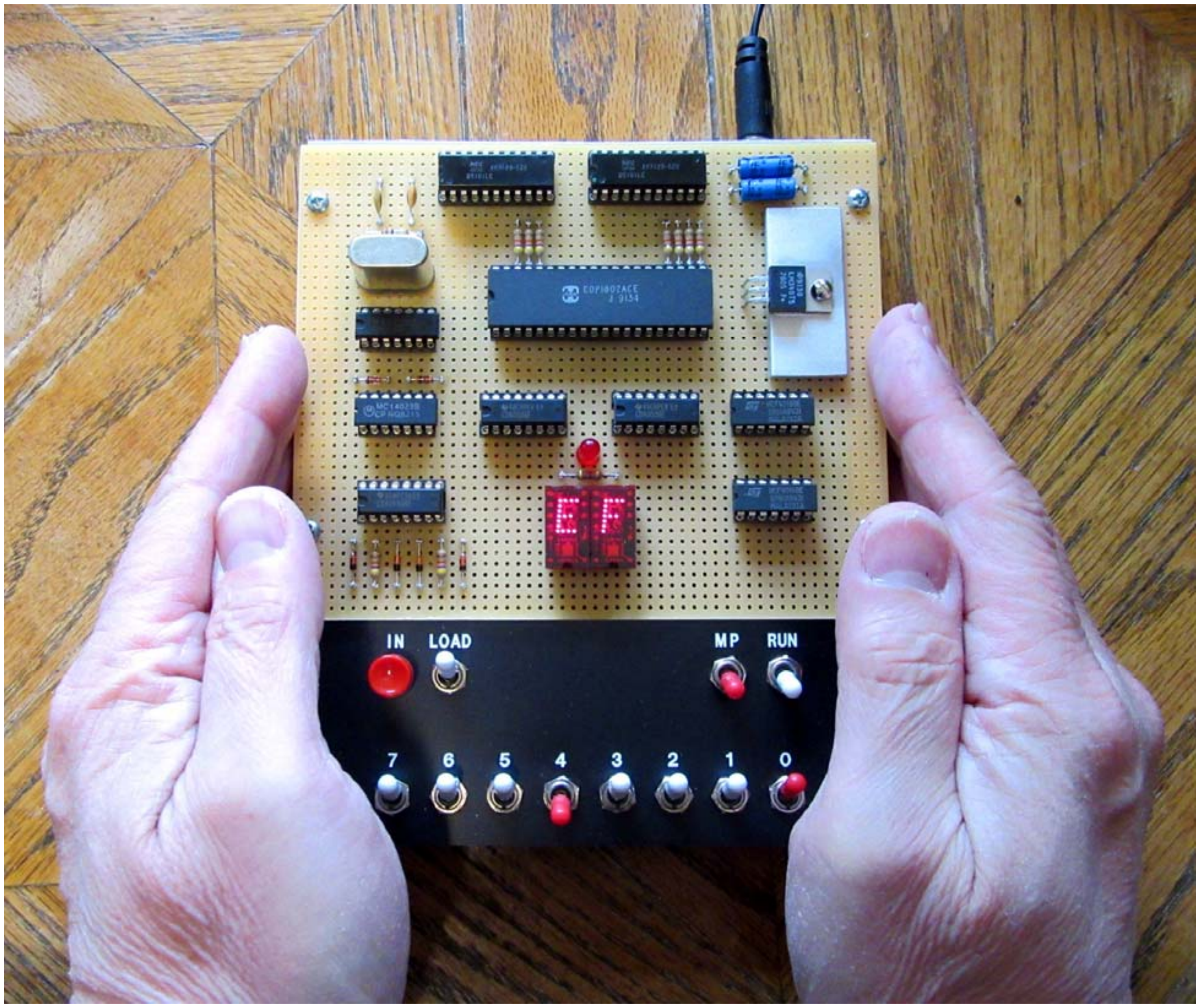
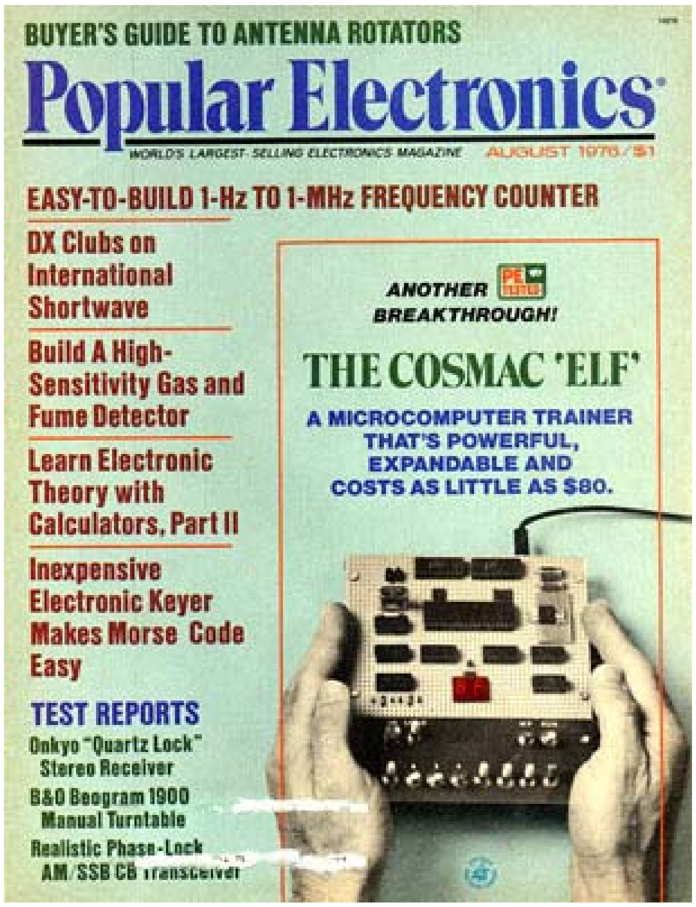


photo of the author's hands holding his ELF build, the basis for this manual and related documents

TABLE OF CONTENTS

Associated documents:
- COSMAC "ELF" Schematic Diagram
- COSMAC "ELF" Layout & Dimensions
- COSMAC "ELF" Parts/Supplier List
- COSMAC "ELF" Switch Panel & IC Labels


BUYER'S GUIDE TO ANTENNA ROTATORS
Popular Electronics
WORLD'S LARGEST-SELLING ELECTRONICS MAGAZINE
AUGUST 1976 / $1
EASY-TO-BUILD 1-Hz TO 1-MHz FREQUENCY COUNTER
DX Clubs on International Shortwave
Build A High-Sensitivity Gas and Fume Detector
Learn Electronic Theory with Calculators, Part II
Inexpensive Electronic Keyer Makes Morse Code Easy
TEST REPORTS
Onkyo "Quartz Lock" Stereo Receiver
B&O Beogram 1900 Manual Turntable
Realistic Phase-Lock AM/SSB CB Transceiver
ANOTHER BREAKTHROUGH!
THE COSMAC 'ELF'
A MICROCOMPUTER TRAINER THAT'S POWERFUL, EXPANDABLE AND COSTS AS LITTLE AS $80.

OVERVIEW

The COSMAC “ELF” was a microcomputer design by Joseph Weisbecker that was featured and announced in the pages of the August 1976 edition of Popular Electronics magazine. Weisbecker used his familiarity with the ELF’s 1802 microprocessor (he was one of that processor’s developers) to design a functioning microcomputer based on the 1802 that used a bare minimum of components. He thus made it attractive to hobbyists who wished to learn and experiment with microprocessors, yet could not afford contemporary microcomputers targeted at non-professionals, such as the MITS Altair 8800 which had appeared about a year and a half earlier on the cover of the same magazine.

Both the Altair and the ELF used early 8-bit microprocessors, and both theoretically had the same potential for expansion and practical usefulness, yet the Altair went on to great commercial popularity and influence (such as practically launching Microsoft), while the ELF was built in great numbers by hobbyists and yet is considered to be much less influential. Certainly a major difference was that the Altair had a defined bus and a practical enclosure which at least made it attractive to businesses and expandable by many companies that offered accessory board for it. Contrast this with the ELF, which was initially available only as a schematic and a list of parts, with the hobbyist left up to his own devices to determine if, and how, the basic design might be made expandable. This fact must have discouraged many potential manufacturers from trying to produce accessories. Still, the ELF was, due to its initial simplicity and low cost, a gateway microcomputer that many people could easily afford to own and experiment with.

Even in the 21st Century, the ELF continues to be built by hobbyists for nostalgic reasons, and because it remains a simple and affordable way to really learn what microprocessors are all about. Contrast the ELF with the various “Arduino” and “Raspberry Pi” microcomputer platforms that offer hobbyists powerful ways of implementing microcomputer BASED projects while teaching them very little about microprocessors in the process. Building and programming an ELF is about as basic as it gets when it comes to learning the rudiments of micro-computing.

This User’s Manual was written as part of the documentation package for a new ‘build’ of the original ELF computer made by hobbyist Paul Schmidt in the year 2017. The goal was to provide all necessary documentation in one place, with a high level of consistency between the various documents, so that any prospective hobbyist could build and use an ELF having only this document package to work from. In this, there was considerable effort made to assure an accurate, reliable and complete schematic diagram, dimension drawing, parts layout drawing, parts list and supplier list, along with this User’s Manual.

HISTORY OF THE 1802

The “1802 microprocessor” was originally known as the RCA CDP1802. It is a large scale integration (LSI) microprocessor (µP) ‘chip’ that is implemented using so-called COSMAC (Complementary Symmetry Monolithic Array Computer) architecture. RCA released the 1802 in early 1976 as the company's first single-chip microprocessor.

Joseph Weisbecker was an RCA engineer who developed a new 8-bit computer architecture in the 1970 ~ 1971 time period, with the results being released as the 2-chip pair of COSMAC 1801U and 1801R in early 1975. In 1976, an RCA team led by engineer Jerry Herzog integrated the two chips into one, the 1802.The 1802 was unique at the time for being fabricated using the CMOS (Complementary Silicon/Metal-oxide Semiconductor) process.

The 1802 has an architecture and feature set that is different from most other 8-bit microprocessors of its vintage. It is ultra-low power, it has no minimum clock speed (functioning all the way down in frequency to nearly 0 Hz, and not being disrupted if the clock *is* 0 Hz), it has an 8-bit address bus that handles high and low order bytes of the 16-bit address in subsequent clock cycles (allowing more IC pins to be used for other more interesting features). It has sixteen 16-bit internal general purpose (scratch-pad) registers, any of which can be designated as the program counter or index register, etc. It has a single bit output port that is easily set or reset and tested using software. In addition to the common interrupt, it has four dedicated single bit input ports that can be tested by software and used to redirect program execution. It handles I/O differently from most other contemporary 8-bit microprocessors, giving them dedicated special purpose machine language commands and special I/O control pins on the IC.

Of particular significance to the ELF design, the 1802 incorporates a built-in DMA controller, which allows direct loading of programs into memory without use of any software, and with only a few logic gates external to the processor itself. This allowed Weisbecker to design the ELF so that program instructions and data could be entered into memory using only a set of eight toggle switches, a pushbutton switch, and a small handful of supporting logic ICs. The DMA design also allows each entered byte to be displayed, which is useful for verification.

Unlike many other early 8-bit microprocessors from the 1970s, the 1802 remains in production in the 21st Century, albeit not by the original manufacturer RCA.

The 1802’s unique characteristics have resulted in its continued use in aircraft (e.g. the Boeing 737), Earth-orbiting satellites, the Hubble Space Telescope, the Magellan Venus Probe, the Galileo space probe, the Space Shuttle’s ‘secure communications system’, as well as numerous embedded applications that include point-of-sale electronics (cash registers, etc.), pay phones in Europe, audio/video equipment, and low power medical equipment (e.g. a heart pacemaker).

HISTORY OF THE ELF

The 1970s was an exciting time for people interested in computers, since a new device called the ‘microprocessor’ had only recently come on the market and the possibilities were numerous. Because the new microprocessor had all the same architectural features of the extant large mainframe computers of that time, engineers could immediately see the potential for a new world of ‘personal’ computers. Now computers could be much smaller and affordable, and they could even be built into other products, and could be made portable.

Nevertheless, at this time there were no commercially available ‘personal’ computers available; very little hardware had been developed yet, and no commercially available software to make the microprocessor *do anything*. Early inroads were made by hobbyists who started to design and construct home-made microcomputers, and they wrote software for them. Soon, entrepreneurs like Ed Roberts at MITS (Altair 8800), Joseph Weisbecker (ELF), Steve Wozniak (Apple), Chuck Peddle (Motorola 6800, MOS Technology 6502, Commodore) and others were coming up with new microcomputer designs for those first microprocessors and personal computers.

Probably to help expose a new generation of enthusiasts and engineers to the new RCA 1802, Joe Weisbecker wrote his famous construction article for the ELF, which appeared as a featured ‘cover story’ on the August 1976 edition of Popular Electronics. It was clearly a hobbyist computer, and was as simple, basic, and minimal as possible. As originally described in those pages, the ELF featured:

- 256 bytes of RAM
- Eight toggle switches and a pushbutton for program data input
- Two toggle switches to select between four operating modes
- One toggle switch to control the writing mode for the RAM (memory protect)
- Two digit hexadecimal data display
- One discrete LED connected to the 1802’s single bit “Q” output

This original ELF was capable of accepting small programs, tediously toggled in by the user, reviewing the entered program to check for errors, and then running the program. It gave the user the ability to correct any program errors, then run, pause and stop the program execution.

The ELF had limited native I/O. The pushbutton normally used for program entry could be monitored by the program and its status acted upon. The running program could control the status of the ‘Q’ LED. Using direct I/O instructions, the program could read the binary data from the eight toggle switches, and could write data to the two digit hexadecimal display. No other I/O was possible without adding more circuitry and components than were described in the magazine article.

Even with the limitations of very small RAM, the absence of ROM or any other firmware, rudimentary program and data entry, and minimal I/O, the ELF was ideal as a small, inexpensive platform for the user to learn about microprocessor hardware, architecture and programming.

In subsequent articles in later editions of Popular Electronics, Weisbecker showed how to inexpensively add features to the basic ELF design:

- Discrete LEDs as an optional alternate to the two digit hexadecimal display
- Hexadecimal keypad
- Battery backup for the RAM
- ROM
- Rudimentary video output
- Additional I/O

Weisbecker showed how to implement a minimalist 'operating system' in the form of a monitor program, and how to produce graphics on the video display. Sample programs were offered to demonstrate programming techniques.

Businesses soon realized that there was a market ELF inspired computers that would be easier for a hobbyist to construct, and they began to offer kits. Netronics offered their ELF II, Quest had their Super ELF. More recently, Spare Time Gizmos has produced a fully expanded ELF that they call the ELF 2000 (or ELF 2k). Retro Technology has the tiny "1802 Membership Card" which is basically the original ELF but with some minor additions to support serial communications and DMA program loading via a PC's parallel port, with the whole computer fitting inside an Altoids tin.

Also shortly after the arrival of the ELF, RCA produced some commercial microcomputers that are closely related to the ELF. The 'Microtutor' was a modular demonstration platform for microprocessors, including the 1802, that closely resembled the ELF's eight toggle switch and two digit display, as well as its simple mode control. The later 'Microtutor II' was basically the same but had the 1802 built in and thus it could not be used with other microprocessors. RCA also produced the COSMAC VIP, which was essentially the fully expanded ELF design from Weisbecker's series of magazine articles, including the hex keypad, video output, a cassette tape interface for saving and loading programs, a monitor program in ROM, and plug-in accessory boards to add interface for a QWERTY keyboard, tiny-BASIC programming language, and various I/O.

The series of documents, of which this user manual is part, is based on the research leading up to the new 'build' of an ELF in early 2017 by Paul Schmidt. The goal was to recreate the original ELF, without the later expansions, as closely as practical. The dimensions, materials, parts, circuit, and appearance were all replicated. Unlike most other contemporary ELF builds, there was no intention to make this ELF "useful" beyond those capabilities that existed in the original; nostalgic accuracy was the goal, within reason! The only exceptions are the optional addition of an audio output driven from the 'Q' line of the 1802, and an optional variable speed oscillator (inspired by what inventor Lee Hart used in his 1802 Membership Card design) that can be used as a slow rate clock in place of the fixed rate 1 MHz crystal. Because the original ELF's large-body 1 MHz crystal is no longer readily available, this ELF design substitutes a hybrid 1 MHz oscillator device; this is authentic since use of such an oscillator is explicitly described in the original ELF article.

ARCHITECTURE OF THE 1802

Although all microprocessors have differences in their internal architecture, the 1802 architecture is more significantly different in many ways. It is both very simple and yet flexible at the same time.

Registers

Without going into extreme detail, the major architectural points are:

- One 8-bit accumulator, called ‘D’
- One 8-bit arithmetic logic unit (ALU); this only works with the ‘D’ accumulator
- Sixteen 16-bit ‘scratch pad’ registers, R0 ~ R9 and RA ~ RF, any of which can be the program counter, index register, etc. Any and all can be operated on in the same way by program instructions, and any can be incremented and decremented by the program logic
- Three 4-bit registers ‘N’, ‘P’, and ‘X’, which hold the binary/hex codes used to select which of the scratch pad registers will be used for certain purposes; e.g. the code in ‘P’ determines which of the scratch pad registers will be the program counter
- One 4-bit register ‘I’ which is immediately set to the value of the most-significant nibble of an instruction when it is read in from memory; because of the way that the 1802’s instruction op-codes are named in hexadecimal, the left (most significant) hex digit of an opcode defines what type of instruction it is, and thus this value is stored in the ‘I’ register and used by the 1802 to determine how the right (least significant) hex digit of the op-code should be interpreted
- One 8-bit temporary register ‘T’ that is used for executing certain instructions
- One 16-bit address register ‘A’ that holds the address so that it can be separated into low order and high order bytes before being sent out on the 8-bit address bus
- One 1-bit register ‘DF’ which is set = 0 when beginning an ADD or SHIFT operation. If the results of the ADD overflow the ‘D’ register (i.e. an overflow or ‘carry’ occurs), then ‘DF’ is set = 1. Similarly if a SHIFT operation results in an overflow/carry, ‘DF’ = 1. This can be tested by the program after doing such an operation to take appropriate action.
- One 1-bit register ‘Q’ which controls the ‘Q’ output of the 1802, and instructions can set, reset, and test this register

Since the scratch-pad registers are not defined in terms of their use or functionality, they may be used for data storage (like a tiny bank of RAM), or for temporary data storage and manipulation while a program is running, as well as for special functions if designated to be the program counter, index register, etc.

Naming Conventions / Notation

All documents referring to the 1802's internal operation and/or instruction set use the hexadecimal numbering system. For the reader who is not familiar with hexadecimal (*hex*), here is a quick overview.

Just like the decimal numbering system that everyone is familiar with, *hex* follows the same conventions and rules, with the exception that instead of being based on 10 possible values for each digit's position in a number, *hex* has 16 possible values for each position. Just as with decimal, where the possible values are called 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, the values in *hex* are called 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, where A = decimal 10, B = decimal 11, and so on up to F = decimal 15. The letters are used because we have no numeric symbols for values greater than 9. In decimal, when a value in a given position exceeds 9, that position resets to 0 and the next highest position is increased by one, e.g. a decimal 109 that has a value of 1 added will roll over to 110, and 108 that has 83 added will roll over to 191. In *hex*, a 019F that has 1 added will roll over to 01A0, and A4 that has D (decimal 13) added will roll over to B1.

Likewise, the binary numbering system works the same way but to the opposite extreme; where *hex* has more possible values for a given numerical position than the numbers 0 ~ 9 can represent (requiring the use of letters for values greater than 9), in binary there are only two possible values for a given numerical position, namely 0 and 1.

While the 1802 uses only *hex* notation in its documentation and instruction set, the ELF uses binary for the eight toggle switches, and *hex* for the two digit hex display. The ELF user must be familiar with both numbering systems; when entering each new program byte, the program listings will be in *hex*, and the user must mentally convert to binary in order to enter via the eight toggle switches, and then recognize the equivalent entered data presented in *hex* on the display.

For convenience, here is a table of decimal, binary and hex values.

DECIMAL	BINARY	HEX
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Since 1802 op-codes and other values are presented in *hex*, and since *hex* can easily be thought of as one 4-bit binary 'nibble' per hex digit, the user can more readily convert between the binary for the ELF's toggle switches, and the *hex* for the program listing and the ELF's display.

For example, a program listing might specify a byte (whether for an 1802 instruction or for program data) as A4. No elaborate conversion is required as it would be going from *hex* to decimal (i.e. A = decimal 10 by itself, but it is in the next numerical position which has a weight of 16 decimal so it actually is 160 in decimal (16 x 10), plus 4 (in either decimal or hex), equals 160 + 4 = 164 decimal……..OR via the binary route where A = 1010 binary, 4 = 0100 binary, or put together 10100100 binary, which according to the weights given each numerical position is (128 x 1) + (64 x 0) + (32 x 1) + (16 x 0) + (8 x 0) + (4 x 1) + (2 x 0) + (1 x 0) = 128 decimal + 32 decimal + 4 decimal = 164 decimal. No, in the case of the ELF, the user would simply divide the hex A4 into an A and a 4, then do the relatively easy-to-remember conversion to binary (you only need to remember, or to count quickly in your head) so A = 1010 and 4 = 0100, then think of the ELF's eight toggle switches as two banks of 4 bits each, and set the left four switches for ON/OFF/ON/OFF and then set the right four switches for OFF/ON/OFF/OFF. After you do a few programs this way, it becomes second nature and can be done 'in your sleep'.

In the 1802, each of the 16-bit scratch-pad registers is designated R0 ~ RF, and since those designations 0 ~ F can be represented in 4 bits of binary, or one digit of *hex*, a simple hex notation system is used in the documentation and instruction set. For example, one way of flexibly designating a register is R(N), where this means that the 1802 will look at the 4-bit value in register 'N', and use that value to determine which of the sixteen 'R' registers is being referred to; e.g. if 'N' currently holds a value of 0011, or 3, then when R(N) is expressed the 1802 will interpret this as meaning R3, or R(3). This is simple for referring to the entire 16-bit registers, but when half of a register is being referred to, notation such as R(3).0 for the low-order (least significant) byte, and R(3).1 for the high-order (most significant) byte, is used.

Notations such as R(X), R(N) and R(P) simply mean those 'R' registers specified by the 4-bit contents of the 'X', 'N' and 'P' registers.

When a notation such as R(N).0 > D is encountered, it means that the 8-bit contents of the low-order byte of the register specified by the contents of register 'N' should be copied to the 8-bit accumulator register 'D'.

All instructions for the 1802 follow a format where the high-order 4 bits (the left or most significant *hex* digit) will be copied into the 'I' register and will be referred to as 'I', and the low-order 4 bits (the right or least significant *hex* digit) will be copied into the 'N' register and will be referred to as 'N'.

The letter 'M' refers to memory. So the notation M(R(P)) > I,N; R(P)+1 describes an example of what happens when the 1802 fetches a byte from a memory location. The 4-bit register 'P' points at a 16-bit register, and whatever 16-bit data is currently in that specified register will be the address of a memory location, and the 1802 will fetch the 8-bit contents of that memory location, and will treat it as an instruction, so the high-order 4-bits of the fetched data will be stored in register 'I' and the low-order 4-bits of the fetched data will be stored in register 'N', then the

1802 will increment the value of the 16-bit word in the specified register that was just used to indicate the memory location for the fetch….because it has been incremented, it automatically points at the next memory location where the next byte of data will be presumably fetched from. This notation system is described here because it is crucial to understanding how the various 1802 instructions operate when viewing the instruction set listing.

This manual cannot practically list every nuance of the instruction set and notation system, but this overview should help. It is recommended that the ELF user obtain any of the used 1802 manuals that are available (on eBay, etc.) or download and print an 1802 manual and/or instruction set guide from the web/internet. However, a short form of the instruction set descriptions appears later in this section.

Addressing Modes

There are four basic ways (modes) of addressing in the 1802 architecture:

- Register
- Register-indirect
- Immediate
- Stack

In ***Register*** addressing, the address of the operand is contained in the low-order, or 'N' section of the instruction. This mode allows the user to directly address any of the sixteen scratch-pad registers for the purpose of counting or moving data in or out of those registers. Examples: Decrement (2N) and GET LOW (8N), where 'N' specified the scratch-pad register that will be affected.

In ***Register-Indirect*** addressing, the 1802's registers are used as pointers to memory. In this mode, rather than the selected register containing data, instead it contains the address of the memory location that holds the data. A 4-bit address in register 'N' will specify one of the sixteen scratch-pad registers, the contents of which are the address of the data in memory.

In ***Immediate*** addressing, the scratch-pad register selected by the contents of register 'R(P)' (in other words, the current program counter register) addresses memory so that the operand is the byte following the instruction. For example, say that the program counter R(P) points at memory location 0013 and an instruction is fetched from that location; the instruction is a two-byte instruction where the operand is expected to be located in the subsequent byte in memory. When executing the instruction, the 1802 automatically increments R(P) to the next memory location 0014, so that on the next cycle the data from that location will be read from memory and used as the operand for the instruction.

Stack addressing mode is not covered here, but may be studied using any of the various 1802 manuals or online references.

THE 1802 INSTRUCTION SET

Compared to most other microprocessors that were its contemporaries, the 1802's instruction set is a model of order and clarity. Some of this order is due to the fact that the 1802 has those sixteen 16-bit scratch-pad registers, and so many instructions and other operations require the use of one or more of those registers. Since there are sixteen of them, it only makes sense that the instruction set be laid out in a matrix where the functions of the instructions are arrayed as rows, and the specifics of the instructions are arrayed as columns; know the type of instruction and which register it affects, and you know the row and column in the matrix where that instruction resides, and hence its hex code. An experienced 1802 programmer can mostly hand code using only a little bit of his ability to recall a few details, and from there he can synthesize most all of the instruction set without looking anything up. On the next page is a table that has been reproduced in several versions, but all illustrate the 1802's orderly instruction set matrix.

In the table, the instructions fall into several categories:

Register Operations
This group includes instruction types that are used to count and to move data between internal 1802 registers.

Memory Reference
This group of instruction types allows for saving data to a memory location, and reading data from a memory location.

Logic Operations
This group of instructions is for performing logic operations such as AND and OR.

Arithmetic Operations
This group of instructions is for performing arithmetic operations such as ADD and SUBTRACT.

Branches
This group of instructions allows for conditional and unconditional branch operations. This includes sixteen short-branch instructions for in-page operation, and eight long-branch instructions for operation anywhere in memory.

Skips
Conditional and unconditional skip instructions allow for both short and long skip operations.

Control
These are instructions to facilitate program interrupt, operand selection, branch and link operations, and control of the built-in 'Q' flip-flop that in turn controls the status of the 'Q' output.

I/O Transfer
Seven instructions to load the 1802 and memory from I/O control circuits, and seven for saving.

1802 INSTRUCTION MATRIX

		‘N’															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	IDL	“LDN” LOAD ‘D’ FROM REGISTER R(N) – EXCEPT R0														
	1	“INC” INCREMENT REGISTER R(N)															
	2	“DEC” DECREMENT REGISTER R(N)															
	3	BR	BRANCH ON Q , Z , F			BRANCH ON EF1 ~ EF4 = 1				SKP	BRANCH NOT ON Q , Z , F			BRANCH ON EF1 ~ EF4 = 0			
	4	“LDA” LOAD ‘D’ FROM ADDRESS IN REGISTER R(N) & ADVANCE															
	5	“STR” STORE ‘D’ INTO ADDRESS POINTED TO BY REGISTER R(N)															
	6	IRX	OUTPUT							-	INPUT						
‘I’	7	CONTROL & MEMORY REF.				ARITHMETIC w/ CARRY				CONTROL		RST/SET ‘Q’		ARITHMETIC, IMMED. w/ CARRY			
	8	“GLO” GET LOW BYTE OF REGISTER R(N), PUT IN ‘D’															
	9	“GHI” GET HIGH BYTE OF REGISTER R(N), PUT IN ‘D’															
	A	“PLO” PUT ‘D’ INTO LOW BYTE OF REGISTER R(N)															
	B	“PHI” PUT ‘D’ INTO HIGH BYTE OF REGISTER R(N)															
	C	LONG BRANCHES				NOP	LONG SKIPS				LONG BRANCHES			LONG SKIPS			
	D	“SEP” SET REGISTER ‘P’ FROM LOWER BYTE OF INSTRUCTION (N)															
	E	“SEX” SET REGISTER ‘X’ FROM LOWER BYTE OF INSTRUCTION (N)															
	F	LOGIC				ARITHMETIC				LOGIC & ARITHMETIC IMMEDIATE				ARITHMETIC IMMED.			

Note that in the table in the preceding page, most instructions are of types that affect all sixteen scratch-pad registers, and hence their sixteen variants span the entire width of the matrix. For example, the INC instruction type, which increments a specified scratch-pad register, has sixteen versions, each affecting a different one of those sixteen registers. The high-order (I) part of the INC instruction is always '1', and the low-order (N) part of the instruction can be any hex value from 0 to F. The version of the INC instruction called '12' is easily identified because its 'I' part is '1' which tells us that it is an INC instruction, and its 'N' part is '2' which means it is scratch-pad register R(2) that will be effected (incremented, in this instance). The LDA instruction similarly resides in matrix row 4 ('I' = 4), and its 'N' part can refer to any of the sixteen registers, 0 ~ F.

Working our way down the matrix, here are general explanations for what each instruction type does. Note that the 'N' part of an instruction is immediately stored in the 'N' register, so when the explanations refer to 'N' on an instruction, that 'N' value comes from the low-order part of the instruction, so both 'N' references are the same thing.

I = 0, N = 0, **IDL**
The 1802 (CPU) repeatedly cycles on the same instruction, waiting for an I/O request (either DMA-IN, DMA-OUT, or INT) are activated. After receiving the interrupt, normal operation resumes.

I = 0, N = 1 ~ F, **LDN**
This is the LOAD VIA N instruction. The CPU looks at the contents of register 'N', which specifies which of the sixteen scratch-pad registers (except R(0) in this instance) will contain the address of a memory location, and then the CPU goes to that memory location, gets the data byte stored there, and saves it in the accumulator ('D' register). If 'N' = 0, then this is not an LDN instruction; rather it is the IDL instruction instead.

I = 1, N = 0 ~ F, **INC**
This is the INCREMENT REGISTER N instruction. The scratch-pad register specified by register 'N', R(N), is incremented. If the specified register contains FFFF and is incremented, it rolls over to 0000. There is no 'overflow' bit in the CPU which would be set in this instance.

I = 2, N = 0 ~ F, **DEC**
This is the DECREMENT REGISTER N instruction. The scratch-pad register specified by register 'N', R(N), is decremented. If the specified register contains 0000 and is decremented, it rolls over to FFFF. There is no 'underflow' bit in the CPU which would be set in this instance.

I = 3, N = 0, **BR**
This is the UNCONDITIONAL SHORT BRANCH instruction which operates by replacing the program counter's low-order (least significant) byte with the data from the byte immediately following this instruction. Remember that the program counter is the scratch-pad register pointed to by the current contents of the 'P' register, and this is R(0) by default. The low-order byte's name is R(P).0. Examples: If the program counter is at 0E1A when the BR instruction is executed, the CPU will look at memory location 0E1B for a data byte. Assume that byte contains 3C. The CPU will overwrite the low-order byte of the program counter with 3C, so it will end up being 0E3C, and the next instruction fetched will be from that memory location.

I = 3, N = 1, **BQ**
This SHORT BRANCH IF Q = 1 instruction operates similarly to the BR instruction described above, except the CPU ignores and skips over this instruction if the status of the 1-bit 'Q' register is NOT = 1. So if 'Q' =1, the branch is implemented, but if 'Q' = 0 the branch is ignored and the next instruction is fetched; the CPU knows to skip over the data byte following the BQ instruction in this case.

I = 3, N = 2, **BZ**
This SHORT BRANCH IF D = 0 instruction operates similarly to the BQ instruction described above, except whether the branch is implemented or not depends on the contents of the 'D' register instead of the 'Q' register. If 'D' = 0 the branch is implemented, and if 'D' is some other value, the CPU skips to the next instruction.

I = 3, N = 3, **BDF (aka BPZ, BGE)**
This SHORT BRANCH IF DF = 1 instruction operates similarly to the BQ instruction described above, except whether the branch is implemented or not depends on the status of the 1-bit 'DF' register instead of the 'Q' register. If 'DF' = 1 the branch is implemented, and if 'D' = 0, the CPU skips to the next instruction. BDF means 'Short branch if DF =1', BPZ means 'Short branch if 'D' is positive or zero', BGE means 'Short branch if equal or greater'.

I = 3, N = 4, **B1**
This SHORT BRANCH IF EF1 = 1 instruction operates similarly to the BQ instruction described above, except whether the branch is implemented or not depends on the status of the 1-bit 'EF1' input pin on the 1802, instead of the 'Q' register. Since EF1 is an inverse acting input, a low (0 Volt, or grounded) signal on this input means that the CPU will regard this input as being logically true, or =1. If 'EF1' = 1 the branch is implemented, and if 'EF1' = 0, the CPU skips to the next instruction.

I = 3, N = 5, 6, 7, **B2, B3, B4**
These branch instructions operate identically to the B1 instruction described above, except they test different pins on the 1802, namely EF2, EF3, and EF4 respectively.

I = 3, N = 8, **SKP (aka NBR)**
This is the SHORT SKIP, or NO SHORT BRANCH instruction. When the CPU encounters this instruction, the address immediately following the instruction is skipped over. In this way, it behaves just like any of the preceding branch instructions, except that it does not test anything to determine if the branch should be implemented or not; this branch (or skip) is ALWAYS implemented. This instruction can be used a placeholder for another two-byte branch instruction.

I = 3, N = 9, **BNQ**
This SHORT BRANCH IF Q = 0 instruction operates identically to the BQ instruction described above, except that it tests for the ‘Q’ register to = 0, so a branch occurs only if ‘Q’ = 0.

I = 3, N = A, **BNZ**
This SHORT BRANCH IF D NOT 0 instruction operates identically to the BZ instruction described above, except the branch is implemented only if the bye in register ‘D’ is NOT equal to 0.

I = 3, N = B, **BNF (aka BM, BL)**
This SHORT BRANCH IF DF = 0, or SHORT BRANCH IF POSITIVE OR ZERO, or SHORT BRANCH IF EQUAL OR GREATER instruction operates identically to the BDF instruction described above, except the branch is implemented if ‘DF’ = 0.

I = 3, N = C, D, E, F, **BN1, BN2, BN3, BN4**
These branch instructions operate identically to the B1 ~ B4 instructions described above, except they test for the EF1 ~ EF4 pins to be logically = 0 (which means the corresponding pins on the 1802 must be at (or near) the same voltage potential as the CPU’s positive power supply, i.e. NOT at 0V / grounded).

I = 4, N = 0 ~ F, **LDA**
This is the LOAD ADVANCE instruction. The scratch-pad register specified by register ‘N’, R(N), contains the address of a memory location; the CPU fetches the data from that location and stores it in the ‘D’ register. Then the CPU increments the contents of R(N).

I = 5, N = 0 ~ F, **STR**
This is the STORE VIA N instruction. The scratch-pad register specified by register ‘N’, R(N), contains the address of a memory location; the CPU fetches the data from the ‘D’ register and stores it in the specified memory location. The contents of ‘D’ are not changed.

I = 6, N = 0, **IRX**
This is the INCREMENT REGISTER X instruction. The scratch-pad register specified by register ‘X’, R(X), is incremented.

I = 6, N = 1, 2, 3, 4, 5, 6, 7, **OUT**
This is the OUTPUT TO I/O instruction. The scratch-pad register specified by register 'X', R(X), contains the address of a memory location. The CPU retrieves the data from that memory location and places it on the data bus. Then the three low-order bits of register 'N' are used to control the status of the three output pins of the 1802 called N0, N1, and N2. These pins may be connected to logic that selects various I/O devices. If the binary pattern on N0 ~ N2 corresponds to a logically gated I/O device (e.g. the eight toggle switches or the 2-digit hex display of the ELF), then the CPU's I/O control system automatically takes care of writing the data on the bus to the logically selected I/O device (it must be a device capable of being written to, since this is an output instruction). Normally, the three N outputs (or any subset of these three outputs) may be gated along with the 1802's MRD output (the MRD pin must be low). The R(X) is automatically incremented after writing the data to the I/O device. If subsequent outputs of data from the same memory location are required, the value in R(X) must be changed back to point to the correct device before executing this instruction again. Note that the 'N' part of this instruction can have a value from 1 ~ 7, and this gets written into the 'N' register before executing the rest of the instruction. Hence, the 'N' part of this instruction influences which I/O device will be written to. Since there is no option for 'N' to be = 0 with this instruction, then the three low-order bits of register 'N' cannot be equal to 0, and thus at least one of them must be = 1. Because of this clever arrangement, at least one of the N outputs will be on during execution of this instruction, and therefore an I/O device will be selected. If this instruction allowed for 'N' to = 0, then none of the N outputs would be on and no I/O device would be selected.

I = 6, N = 8 (no instruction)
This is the only position on the instruction matrix which does not have a corresponding instruction.

I = 6, N = 9, A, B, C, D, E, F, **INP**
This is the INPUT FROM I/O instruction. This instruction works similarly to the OUT instruction described above, but in reverse, reading a byte of data from an I/O device. The scratch-pad register specified by register 'X', R(X), contains the address of a memory location. The 'N' part of the instruction is 9 ~ F, and this is written in the 'N' register. Because the I/O system only uses the three low-order bits of the 'N' register, the most significant bit is ignored. As a result, if 9 is written into 'N', the I/O system sees 1 (9 = 1001 binary, and if dropping the most significant bit the binary number becomes 0001, or 1). Similarly a written A is read by the I/O system as 2, etc; up through a written F being read as 7. In this way, this instruction corresponds to the OUT instruction's 'N' part which can be 1 ~ 7. Normally, the three N outputs (or any subset of these three outputs) may be gated along with the 1802's MRD output (the MRD pin must be high). Then the CPU's I/O control system automatically takes care of reading the data from the selected I/O device (it must be a device capable of being read from, since this is an input instruction). The data is written into the memory location specified by R(X), and the same data is also saved to the 'D' register. The R(X) register is NOT incremented.

I = 7, N = 0, **RET**
This is the RETURN instruction. The single hex digits in the 'X' and 'P' registers are replaced by the data stored in the memory byte addressed by scratch-pad register R(X), and R(X) is then incremented. The 1-bit 'IE' register (Interrupt Enable) is set = 1. Even if the other aspects of this instruction are not required, it can be used solely to set the 'IE' bit.

I = 7, N = 1, **DIS**
This is the DISABLE instruction, and it is similar to the RET instruction described above, except instead of setting 'IE' = 1, 'IE' is reset to = 0. Even if the other aspects of this instruction are not required, it can be used solely to reset the 'IE' bit.

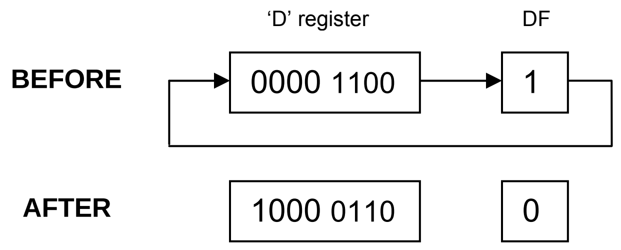
I = 7, N = 2, **LDXA**
This is the LOAD 'D' VIA 'X' AND ADVANCE instruction. The data in the memory location addressed by the scratch-pad register specified by 'X', R(X), is placed into register 'D', and the address in R(X) is incremented. The contents of the memory location specified by R(X) are not changed.

I = 7, N = 3, **STXD**
This is the STORE 'D' VIA 'X' AND DECREMENT instruction. The byte in 'D' is stored in the memory location addressed by the contents of the register specified by 'X', R(X), and R(X) is decremented.

I = 7, N = 4, **ADC**
This is the ADD WITH CARRY instruction. The memory byte addressed by the scratch-pad register specified by 'X', R(X) is added to the contents of the 'D' register, and also the contents of the 1-bit 'DF' register is also added to 'D'. The 8-bit result of the double addition is stored in 'D'. Regardless of what the status of 'DF' was before the operation, afterwards 'DF' will be set = 1 if the addition resulted in a carry, and if not it will be reset = 0.

I = 7, N = 5, **SDB**
This is the SUBTRACT WITH BORROW instruction. The value of the 'D' register is subtracted from the memory byte addressed by the scratch-pad register specified by 'X', R(X), and then the inverted contents of the 1-bit 'DF' register is also subtracted from the same memory location, and the result of the double subtraction is stored in the 'D' register. The 'DF' is also changed if necessary as a result of the subtraction.

I = 7, N = 6, **SHRC (aka RSHR)**
This is the SHIFT RIGHT WITH CARRY, or RING SHIFT RIGHT instruction. The binary contents of the 'D' register are shifted one bit position to the right, or least significant direction. The '0', or lowest order, position of the byte is moved to the CARRY bit 'DF', while the previous contents of 'DF' are moved to the '7', or highest order, position of the 'D' register.



I = 7, N = 7, **SMB**
This is the SUBTRACT MEMORY WITH BORROW instruction. The byte in memory addressed by R(X), plus the borrow (indicated by register 'DF = 0) is subtracted from the 'D' register. The result is placed in the 'D' register, and if a new borrow occurred as a result of the operation, 'DF' will be reset to = 0. This instruction is similar to SDB described above, but with the operands ('D' and the memory location) reversed.

I = 7, N = 8, **SAV**
This is the SAVE instruction. The byte contained in the 8-bit 'T' register is stored at the memory location addressed by the scratch-pad register specified by 'X', R(X). Subsequent execution of a RETURN (RET) or DISABLE (DIS) instruction, as described above, can then replace the original 'X' and 'P' values to resume, or return to, normal program execution.

I = 7, N = 9, **MARK**
This is the PUSH X, P TO STACK instruction. The current contents of the two 4-bit registers 'X' and 'P' are combined as 'XP' and stored in the 'T' register, and the same 8-bit value is also stored in the memory location addressed by scratch-pad register R(2). The contents of 'P' are then stored in 'X' and the 16-bit value in R(2) is decremented.

I = 7, N = A, **REQ**
This is the RESET 'Q' instruction. The contents of the 1-bit register 'Q' is reset = 0, and since 'Q' controls the 'Q' output of the 1802, that output goes low (to 0V or circuit ground potential).

I = 7, N = B, **SEQ**
This is the SET 'Q' instruction. The contents of the 1-bit register 'Q' is set = 1, and since 'Q' controls the 'Q' output of the 1802, that output goes high (to the same potential as the power supply voltage to the 1802 IC).

I = 7, N = C, **ADCI**
This is the ADD WITH CARRY - IMMEDIATE instruction. The contents of the memory location addressed by the scratch-pad register specified by ‘P’, R(P), is added to the contents of the ‘D’ register, and the results are stored in the ‘D’ register. The final state of the ‘DF’ register indicated whether or not a carry occurred as a result of the addition. Register R(P) is incremented.

I = 7, N = D, **SDBI**
This is the SUBTRACT D WITH BORROW - IMMEDIATE instruction. Similar to the SBD instruction described above, except instead of the minuend coming from an addressed memory location, it comes from the byte immediately following this instruction.

I = 7, N = E, **SHLC (aka RSHL)**
This is the SHIFT LEFT WITH CARRY, or RING SHIFT LEFT instruction. It is similar to the SHRC instruction described above, except the data is shifted to the left instead of to the right. The contents of ‘DF’ are moved into the least significant bit of ‘D’, and the most significant bit of ‘D’ is moved into ‘DF’.

I = 7, N = F, **SMBI**
This is the SUBTRACT MEMORY WITH BORROW - IMMEDIATE instruction. It is similar to the SMB function described above, except instead of the subtrahend coming from an addressed byte in memory, it comes from the byte immediately following the instruction.

I = 8, N = 0 ~ F, **GLO**
This is the GET LOW REG N instruction. The low-order byte of the scratch-pad register specified by ‘N’ is placed on the ‘D’ register.

I = 9, N = 0 ~ F, **GHI**
This is the GET HIGH REG N instruction. The high-order byte of the scratch-pad register specified by ‘N’ is placed on the ‘D’ register.

I = A, N = 0 ~ F, **PLO**
This is the PUT LOW REG N instruction. The data in the ‘D’ register is copied to the low-order byte of the scratch-pad register specified by ‘N’. The contents of ‘D’ are not changed.

I = B, N = 0 ~ F, **PHI**
This is the PUT HIGH REG N instruction. The data in the ‘D’ register is copied to the high-order byte of the scratch-pad register specified by ‘N’. The contents of ‘D’ are not changed.

I = C, N = 0, **LBR**
This is the unconditional LONG BRANCH instruction. The two bytes of data following the instruction are copied and then written into the full 16-bit scratch-pad register specified by ‘P’, R(P). For example, if the program contains these three bytes in sequence: C0 25 3A, then the contents of R(P) will be 253A, which is treated as an address. The CPU will execute its next instruction at that address, jumping forward or backwards as required to do so.

I = C, N = 1, **LBQ**
This is the LONG BRANCH IF Q instruction. It is similar to the LBR instruction above, except it only implements the branch if the ‘Q’ register = 1.

I = C, N = 2, **LBZ**
This is the LONG BRANCH IF D = 0 instruction. It is similar to the LBQ instruction above, except it only implements the branch if the contents of the ‘D’ register = 0.

I = C, N = 3, **LBDF**
This is the LONG BRANCH IF DF = 1 instruction. It is similar to the LBQ instruction above, except it only implements the branch if the contents of the ‘DF’ register = 1.

I = C, N = 4, **NOP**
This is the NO OPERATION instruction. The CPU does nothing with this instruction, and executes the instruction at the next address.

I = C, N = 5, **LSNQ**
This is the LONG SKIP IF Q = 0 instruction. If ‘Q’ = 0, the next two bytes following this instruction are skipped. For example, if the following bytes appear in sequence: C5 55 25 F2, and the C5 is executed, the CPU will check to see if Q = 0. If it is, then the next instruction executed will be the F2. If it is not, then the next instruction executed will be the 55.

I = C, N = 6, **LSNZ**
This is the LONG SKIP IF D IS NOT 0 instruction. It works like the LSNQ instruction above, except the test is for the ‘D’ register being any value other than 0.

I = C, N = 7, **LSNF**
This is the LONG SKIP IF DF = 0 instruction. It works like the LSNQ instruction above, except the test is for the ‘DF’ register being = 0.

I = C, N = 8, **LSKP (aka NLBR)**
This is the unconditional LONG SKIP, or NO LONG BRANCH instruction. The two bytes following the C8 instruction are skipped and the CPU executes the instruction that follows the skipped bytes. The alternate instruction mnemonic implies that the two skipped bytes represent an unused branch address.

I = C, N = 9, **LBNQ**
This is the LONG BRANCH IF Q = 0 instruction. It is similar to the LBQ instruction above, except it only implements the branch if the ‘Q’ register = 0.

I = C, N = A, **LBNZ**
This is the LONG BRANCH IF D NOT 0 instruction. It is similar to the LBZ instruction above, except it only implements the branch if the contents of the ‘D’ register is some value other than 0.

I = C, N = B, **LBNF**
This is the LONG BRANCH IF DF = 0 instruction. It is similar to the LBDF instruction above, except it only implements the branch if the contents of the 'DF' register = 0.

I = C, N = C, **LSIE**
This is the LONG SKIP IF IE = 1 instruction. It works like the LSNQ instruction above, except the test is for the 1-bit 'IE' register being = 1.

I = C, N = D, **LSQ**
This is the LONG SKIP IF Q = 1 instruction. It works like the LSNQ instruction above, except the test is for the 'Q' register being = 1.

I = C, N = E, **LSZ**
This is the LONG SKIP IF D = 0 instruction. It works like the LSNZ instruction above, except the test is for the 'D' register being = 0.

I = C, N = F, **LSDF**
This is the LONG SKIP IF DF = 1 instruction. It works like the LSNF instruction above, except the test is for the 'DF' register being = 1.

I = D, N = 0 ~ F, **SEP**
This is the SET P instruction. The single hex digit (4-bit binary) currently in the 'N' register is copied to the 'P' register. This is used to specify which of the scratch-pad registers will be used as the program counter (remember that R(P) is the defined program counter). When 'P' is set by this instruction, the CPU immediately jumps to the instruction sequence beginning at the memory address stored in the scratch-pad register specified originally by 'N'.

I = E, N = 0 ~ F, **SEX**
This is the SET X instruction. The single hex digit (4-bit binary) currently in the 'N' register is copied to the 'X' register. This is used to designate R(X) for arithmetic instructions and I/O byte transfer operations.

I = F, N = 0, **LDX**
This is the LOAD VIA X instruction. The contents of the memory byte addressed by the contents of the scratch-pad register specified by 'X', R(X), will be written to the 'D' register. This instruction does not increment the address in the way that the LDA instruction does. The contents of the memory location are not changed.

I = F, N = 1, **OR**
This is the OR instruction. The individual bits of the two 8-bit operands are combined according to the rules for logical OR. The byte currently in the 'D' register is one of the operands, while the contents of the byte in the memory location addressed by R(X) is the second operation. The results of the OR operation are stored in the 'D' register, replacing that operand. This instruction is particularly useful in setting individual bits in a byte of data.

I = F, N = 2, **AND**
This is the AND instruction. It works similarly to the OR instruction described above, except the logical operation is AND. This instruction is particularly useful to test if one or more particular bits in a byte of data are currently = 1, as well as to mask individual bits.

I = F, N = 3, **XOR**
This is the EXCLUSIVE OR instruction. It works similarly to the OR instruction described above, except the logical operation is XOR. This instruction is particularly useful to compare two data bytes for equality since identical values will result in all 0's in 'D'.

I = F, N = 4, **ADD**
This is the ADD instruction. It works similarly to the OR instruction described above, except instead of a logical operation on the two operands, an addition is done. The result is stored in 'D', and 'DF' =1 if a carry occurred, and 'DF' = 0 otherwise.

I = F, N = 5, **SD**
This is the SUBTRACT D instruction. The byte in 'D' is subtracted from the data in the memory location addressed by R(X). The 8-bit result is stored in the 'D' register, replacing the subtrahend. The 'DF' register is modified as a result of this operation; 'DF' = 0 if there was a borrow, and 'DF' = 1 is there was no borrow.

I = F, N = 6, **SHR**
This is the SHIFT RIGHT instruction. It is similar to the SHRC instruction described on page 18, except nothing ('DF' or any other source) is shifted into the most significant bit of 'D'; that bit will always be 0 after this instruction has executed.

I = F, N = 7, **SM**
This is the SUBTRACT MEMORY instruction. It is the opposite of the SD instruction. The byte of data in the memory location addressed by R(X) is subtracted from the data byte currently in 'D'. The 8-bit result is stored in the 'D' register, replacing the minuend. The 'DF' register is modified as a result of this operation; 'DF' = 0 if there was a borrow, and 'DF' = 1 is there was no borrow.

I = F, N = 8, **LDI**
This is the LOAD IMMEDIATE instruction. The data byte immediately following the current instruction's byte will be copied to the 'D' register. The program counter R(P) will be incremented such that it points at the memory location following the data byte, so it is ready for the next instruction fetch.

I = F, N = 9, **ORI**
This is the OR IMMEDIATE instruction. It works similarly to the OR instruction described above, except that the second operand is in the byte immediately following this instruction instead of coming from an addressed memory location.

I = F, N = A, **ANI**
This is the AND IMMEDIATE instruction. It works similarly to the AND instruction described above, except that the second operand is in the byte immediately following this instruction instead of coming from an addressed memory location.

I = F, N = B, **XRI**
This is the EXCLUSIVE OR IMMEDIATE instruction. It works similarly to the XOR instruction described above, except that the second operand is in the byte immediately following this instruction instead of coming from an addressed memory location.

I = F, N = C, **ADI**
This is the ADD IMMEDIATE instruction. It works similarly to the ADD instruction described above, except that the second operand is in the byte immediately following this instruction instead of coming from an addressed memory location.

I = F, N = D, **SDI**
This is the SUBTRACT IMMEDIATE instruction. It works similarly to the SD instruction described above, except that the second operand is in the byte immediately following this instruction instead of coming from an addressed memory location. It is similar to the SMI instruction below except that the operands are reversed.

I = F, N = E, **SHL**
This is the SHIFT LEFT instruction. This works similarly to SHR except the bits are shifted to the left instead of to the right. The final low order bit of ‘D’ will always be 0 after this instruction has executed.

I = F, N = F, **SMI**
This is the SUBTRACT MEMORY IMMEDIATE instruction. It works similarly to the SM instruction described above, except that the second operand is in the byte immediately following this instruction instead of coming from an addressed memory location. It is similar to the SDI instruction above except that the operands are reversed.

ELF CIRCUIT DESCRIPTION

When reading the following, refer to the schematic diagram that is part of this documentation package.

Power Supply

An external regulated 9VDC power source is assumed, according to the schematic and the parts list. The 9V power enters the ELF via jack J1. The power passes through two diodes D8 & D9, each of which drops about 0.7 volts, so the voltage after D9 is about 7.6V. This diode drop is used to reduce the voltage applied to voltage regulator IC13, which needs its input voltage to be at least 2V higher than its output voltage of 5V in order to regulate properly. Any excess input voltage (> 7V) is simply dissipated by IC13 as heat, and with the small heatsink it is desirable to minimize heat production, so the dual diode voltage drops are there solely for the purpose of keeping IC13 as cool as possible. Since the difference between 7.6V and the 7V that IC13 needs to work properly is not a very great value, it is important that the power supply voltage coming in at J1 never goes less than 9V, which is why using a regulated power supply before J1 is critical (an unregulated supply might very well dip below 9V, and this could cause IC13 to stop regulating, which would compromise the final 5V power supply used throughout the ELF circuit). If in doubt, or if you notice that the output of IC13 is less than 5V (allow 4.9V), try removing (or shorting) one of those two diodes to increase the IC13 input voltage by 0.7V.

Capacitor C3 is a filter for the incoming power, and provides a certain amount of ‘boost’ to cover transient demands for current that the ELF circuit may make, but which the external power supply cannot immediately provide due to its own response lag or due to resistance in the long external power supply cable. IC13 requires a small capacitance at its output to allow for stable operation, and C4 provides that stability. The regulated 5V at the output of IC13 (and across C4) is used throughout the ELF circuit.

NOTE: The ELF mostly uses CMOS IC’s, with the exception of the RAM (which will also be CMOS 5101 IC’s unless you choose to use the older 2101 type RAM IC’s), and the two displays IC11 and IC12 which are TTL types. CMOS IC’s are not sensitive to the exact power supply voltage, whereas TTL IC’s are sensitive in the sense that they require that their power supply be very close to 5V. If it were not for the non-CMOS IC’s in the ELF circuit, the whole external power supply and IC13 voltage regulator circuit could be dispensed with, and the circuit could run nicely from an unregulated battery or other power source.

Preferably, the 5V power from voltage regulator IC13 should be distributed to the ELF circuit by a ‘star topology’ wiring arrangement, where the +5V output of IC13 is brought to a ‘bus bar’ and the ground pin of IC13 is brought to a second ‘bus bar’. These bus bars can be simply two short lengths of fairly stiff solid wire, perhaps between 1” and 2” long each. Then, for each IC (or other component that needs to be connected to the power), run a wire from the IC’s V+ pin(s) directly to the 5V bus bar and solder it there, and run another wire from the IC’s ground pin(s) directly to the ground bus bar and solder it there. Do likewise for the power wiring for every IC. In the case of non-power pins on a given IC that are shown on the schematic as being connected to 5V or ground, just wire wrap a short wire from that pin, or those pins, to the appropriate power pin of the same IC; there is no need to run THOSE wires separately back to the bus bars. For the toggle switches and pushbutton switch, each switch has a 5V connection and a ground connection, so you connect those together at the switch panel using wire wrap wire or bus bar wire, and then run a single 5V and ground wire pair back to the IC13 bus bars.

For non-IC components that need to be connected to 5V or ground, e.g. all the pull-up resistors such as R2 through R8, simply wire wrap those together and then run a single wire over to the appropriate IC13 bus bar.

NOTE: The schematic shows certain UNUSED IC inputs connected to ground, e.g. several pins on IC3, IC6, IC7, and IC14. It is critical to do this; CMOS logic device inputs are very high impedance and if left unconnected, or floating, they will pick up electrostatic energy and try to use those as logic signals, and potentially cause problems in the IC(s). Tying unused input pins to ground eliminates this concern. Unused IC outputs can be left unconnected.

Oscillator

The 1802 requires an oscillator in order to function; the oscillator provides the *Clock* signal via IC4 pin 1. The 1802 also has provision for using its internal oscillator, but that requires several external components to work; a crystal (typically 1MHz), a resistor (typically 1M), and two small value capacitors (typically 30pF or 33pF). If using the 1802's internal oscillator, do NOT use the oscillator circuit shown in the schematic, but instead connect the crystal and resistor in parallel between IC4 pins 1 & 39, and connect a 30/33pF capacitor from IC4 pin 1 to ground, and the other 30/33pF capacitor from IC4 pin 39 to ground. IC4 pin 39 is technically an output from the 1802, which is why it does not need to be connected to ground if not being used.

Assuming that the 1802's internal oscillator is not being used, the external oscillator can take a couple of forms according to the schematic:

1) A hybrid oscillator device such as X2, which has an internal crystal and support circuitry to operate as an oscillator.
2) An oscillator circuit consisting of two Schmitt-trigger type inverters (parts of IC14), ceramic resonator component X3 (which has integral capacitors), and a 1M resistor R12. As shown, R12 is actually a trim potentiometer that is connected such that it can be used to vary the oscillator frequency between the specified maximum of the ceramic resonator and a much lower frequency. This is useful for some experiments where it might be nice to be able to adjust the 1802's clock frequency while a program is running, in order to observe the results. Also, when using the ELF to generate a frequency output via 'Q', the variable nature of R12 can come in handy. The Schmitt-trigger type of IC is critical, since the rest of the ceramic resonator will not function with a regular CMOS inverter IC. A 1.8MHz ceramic resonator is suggested, but a slightly different value may be used, as long as the clock signal sent to IC4 pin1 in not higher than about 2MHz; 1MHz is recommended for the ELF.

As shown on the schematic, both the X2 and X3 based oscillators are present, and the user can select between them using a shunt jumper, or an SPDT switch of some kind.

Control Circuit

In the following, ‘1802’ and ‘CPU’ are used interchangeably.’1802’ usually refers to IC pins, while ‘CPU’ usually refers to internal IC functionality.

Defining the 1802’s pins

The 1802 handles a lot of things internally that many of its contemporaries needed many external ICs to accomplish. Of great importance for the ELF is the 1802’s method for implementing Direct Memory Access, or DMA; this is covered in more detail below. The control circuit used for the ELF is rather ingenious in its clever arrangement of circuitry to minimize parts count and cost.

The 1802 IC has an 8-bit Data Bus (pins 8 ~ 15), and an 8-bit Address Bus (pins 25 ~ 32); even though the 1802 uses 16-bit addressing, it multiplexes using the 8-bit Address Bus for both the low order and high order address bytes. The 1802 has two special output pins TPA (pin 34) and TPB (pin 33). When the 1802 is placing the high order byte of a memory address onto the 8-bit Address Bus, the TPA output is pulsed so that memory devices can latch in, or otherwise react, to that part of the address. When the 1802 is placing the low order byte of a memory address onto the 8-bit Address Bus, the TPB output is pulsed in a similar manner. Since only low order memory addresses are used on the ELF, TPA is left unconnected.

The 1802 has five dedicated special purpose 1-bit I/O lines; the four 1-bit inputs EF1 ~ EF4 (pins 21 ~ 24), and the 1-bit output ‘Q’ (pin 4). The ‘EF’ inputs are directly testable using instructions, and two dedicated instructions can set and reset ‘Q’ as if it were a flip-flop with its ‘Q’ output connected to pin 4 (which is basically what ‘Q’ is, and probably why it has this name). The ‘EF’ inputs are all negative acting types, so they are logically on/true when they are pulled to 0V’, and when they are at or near 5V, they are logically off/false.

The 1802 has a dedicated INT input (pin 36) for normal interrupt purposes, just like any other microprocessor. INT is a negative acting input. The ELF does not use this input.

For DMA operations, the 1802 has two dedicated inputs, DMA IN (pin 38) and DMA OUT (pin 37). Both are negative acting inputs. The ELF only uses the DMA IN input.

The CPU has four ‘states’:

- S0 - Fetching an instruction
- S1 - Executing an instruction
- S2 - Responding to a DMA request (via the DMA IN or DMA OUT inputs)
- S3 - Responding to an interrupt request (via the INT input)

To synchronize external devices with the state that the CPU is currently in, the 1802 has two dedicated output lines SC0 and SC1. The four CPU states are coded in 2-digit binary according to their state code numbers 0 ~ 3, and those two binary bits are used to control the SC0 and SC1 outputs. For example, when the CPU is fetching an instruction, this is State 0, or S0, and the binary pattern for ‘0’ is 00, so both the SC0 and SC1 outputs will be 0. When the CPU is handling a DMA request, this is State 2, or S2, and the binary pattern for ‘2’ is 10, so the SC0 output will be 0 and the SC1 output will be 1. The ELF uses only the SC1 output.

The 1802 has a dedicated CLEAR input (pin 3), and this is negative acting. When CLEAR is activated (0V) the CPU is initialized, setting various registers to default states, defining register R0 as the program counter, and resetting R0 to 0000. When CLEAR has been activated but is then returned to its non-active state (5V), this transition starts program execution (the CPU is in RUN mode).

The 1802 has a dedicated WAIT input (pin 2), and this is negative acting. When WAIT is active (0V) the CPU operation is cleanly suspended. If both the CLEAR and WAIT inputs are asserted together, the CPU is put into program LOAD mode.

The 1802 has a special mode for handling dedicated I/O devices, and these are controlled by instructions 61 ~ 67 and 69 ~ 6F (see instruction descriptions in the previous chapter). Seven input devices and seven output devices are allowed for by these instructions. Recalling that the 1802’s instructions are organized by the upper order nibble ‘I’ and the lower order nibble ‘N’, it is the ‘N’ part of these special I/O instructions that define the I/O device to be communicated with. The highest order bit of the ‘N’ value determines if the instruction is for an INPUT device (the bit = 0) or an OUTPUT device (the bit = 1). Accordingly, the lowest order three bits of ‘N’ are what defines the I/O device number, while the highest order bit defines INPUT versus OUTPUT. With only three binary bits defining the I/O device, obviously there can be only eight possibilities (000 ~ 111 binary) for INPUT and eight possibilities for OUT. For both inputs and outputs, the combination when the three bits are all zeroes (000) is NOT allowed for selection of I/O; instruction 60 (0110 0000) is for IRX and 68 (0110 1000) is the only unused function in the 1802. This leaves only seven possibilities (001 ~ 111) for inputs or for outputs, so 61 refers to an INPUT related to I/O device 1 (0001) and 6F refers to an OUTPUT related to I/O device 7 (1111). The state of these three bits is output as N0, N1, and N2 via 1802 pins 17 ~ 19. For example, if instructions 64, 65, 66, 67 or 6C, 6D, 6E, 6F are executed, the most significant of the three bits will be = 1, and thus it is a certainty the N2 output will be on (5V). Using these three outputs, the 1802 can control & select which I/O device it is communicating with. The ELF uses only the N2 output.

The 1802 has two dedicated outputs for interface with memory, MRD (pin 7) and MWR (pin 35); both of these outputs are negative acting, so they will be low (0V) when logically true. When the MRD output is active (0V), this means that the CPU is in a memory READ cycle, and therefore any memory devices need to have their outputs to the Data Bus turned on (i.e. NOT in a tri-state, or high impedance, condition where they would have no effect on the Data Bus). The MWR output is active (0V) when the CPU wishes to write the data that is currently on the Data Bus into the currently addressed memory location.

How the ELF Circuit works

There are eight data toggle switches and three control toggle switches and one control pushbutton switch on the ELF. During normal operation, only two of the control toggle switches and the control pushbutton switch will be moved in ways that could be 'confusing' to the internals of the 1802 or the ELF control circuit in general. This can happen because it is the nature of mechanical switches, whether toggle or pushbutton types, to have their contacts 'bounce' during changes from one position or state to the other position or state. For example, when a pushbutton switch is pressed, its normally closed contact opens fairly cleanly, followed by its normally open contacts closing noisily; the formerly open contacts sort of slam together under the snap action of the switch mechanism, and when the contact points first meet they will hit hard and then bounce apart, then the spring action will force them closed again, and they will bounce open again (although not as far as the first time), and overall there may be several bounces before the switch contacts finally settle down and remain firmly closed.

Many pieces of logic on the ELF control circuit and/or the 1802's input pins are of a type that takes action on transitions from false to true or true to false (off to on, on to off, or 0 to 1, 1 to 0), and when a switch contact initially closes the logic or 1802 input will see a 1, but when the contact bounces open the logic or 1802 input is temporarily not connected to anything, and an 'open' CMOS input tends to float and might be seen as off or on until the switch contacts close again, and so on for as long as the contacts continue to bounce. So instead of a pushbutton press looking to a logic or 1802 input like a simple transition from 0 to 1, it might actually look like 0 1 0 1 0 1 0 1 0 1. This contact bounce can cause undesired circuit operations, and would likely render the ELF unusable in a practical sense. What is needed is a circuit to 'debounce' the switch inputs.

A simple and reliable way to debounce a switch using CMOS logic is to use the method shown on the schematic for the RUN toggle switch SW4, at IC7 pins 14 & 15. This IC segment is a non-inverting buffer, so its output is always the same logical state as its input, only with more power. The common pin of toggle switch SW4 is connected to the input of the buffer (pin 14), so when SW4 is in the UP position the switch common will be tied to 5V and hence the buffer input will be = 1. Since the buffer's output immediately does what the input does, it also goes to 1. *Note that the buffer output is wrapped around so it connects to the input, right where the SW4 common pin is connected.* So now both the switch and the buffer output are pulling the buffer input to 1. When the SW4 contacts bounce open for the first time, the switch is no longer controlling the buffer's input, but the input is not floating to some random value because it is still being pulled to 1 by its output. When the SW4 contacts return from their bounce and are 1 again, this signal agrees with the signal that is already on the buffer input (due to the buffer output) so nothing changes. The switch has thus been debounced, and no matter how many times the contacts bounce open, the buffer output remains = 1. When SW4 is moved back to the DOWN (DN) position, the switch common is now firmly pulled to 0V, and this forces the buffer input to = 0, even though the buffer output is still trying to pull the buffer input = 1. But within a tiny fraction of a second, the buffer output responds to its input being forced to 0, and the output also becomes 0, and thus now the input and output agree with each other and any bouncing of the switch contacts are ignored as they were when the switch was transitioned to its UP position. The downside to this scheme is that the buffer's output circuit is very briefly 'short circuited'. The buffer circuit can tolerate this brief abuse. The output of the buffer thus delivers a debounced version of the SW4 signal to the 1802's CLEAR input (pin 3).

The IN pushbutton SW1 and the LOAD toggle switch SW2 also require debouncing. In addition, the IN pushbutton needs to have its signal inverted before use, and the LOAD switch needs to have both regular and inverted versions of its signal available to logic and the 1802. In both cases, the debounce function is combined with the inversion function by the use of two inverters for each debounce circuit. Two inverters connected end to end will act logically like the buffer did for the RUN switch. In the case of the IN pushbutton, the logic taps off the inverted version of the signal, while in the case of the LOAD switch the inverted version of the signal is used to control the IC3 flip-flop as well as the 1802's WAIT input (pin 2), and the non-inverted version of the signal is used to control the 'N2/LOAD' signal going to IC5.

With the switch debounce understood, let's take a look at the IN pushbutton (SW1) circuit. Normally, the circuit node between IC9 pins 10 & 11 (IN) will = 1. When SW1 is pressed, its common will transition from 0V to 5V, and the IC9 junction will transition from 1 to 0 in response. Any time that SW1 is pressed, and the IC9 junction is 0, this pulls the 1802's EF4 input low (it is normally pulled high, or = 1, by resistor R8) through diode D1, and thus a program can read the IN pushbutton status. Whenever SW1 is pressed and released, the transition during release causes the clock input of flip-flop IC3 (pin 11) to transition from 0 to 1, and this will clock into the flip-flop whatever signal is on the D input (pin 9); since D is always pulled high (= 1), a 1 is always clocked into the flip-flop. This sets the flip-flop's unused Q output (pin 13) = 1 and the used Q-NOT output (pin 12) = 0, which pulls the 1802's DMA IN input low through diode D2 (R2 usually pulls it high). Since DMA IN is a negative acting input, it is thus activated and the CPU responds by immediately reading the data from the eight toggle switches, and storing that data into the currently addressed memory location. As soon as the CPU enters its 'DMA servicing' state, or S2, this means that the higher order of the two State bits will be set and thus the 1802's SC1 output (pin 5) will turn on. This turns the flip-flop's RESET input (pin 10) on via diode D4, and thus the flip-flop is reset and its Q output = 0 and the Q-NOT output = 1, so it stops pulling the 1802's DMA IN input low, and thus nothing is asking the CPU to do a DMA any longer, and when it finishes the current memory save operation (saving the toggle switch data to memory), the CPU returns to normal operation. Of course, this whole cycle with the flip-flop has significance when the CPU is in LOAD mode and is expecting to assist in loading memory directly from an I/O device (the eight toggle switches in the case of the ELF). So, the IN pushbutton circuit works in two ways, to help load memory from the toggle switches, and also to be read by a running program via the EF4 input of the 1802.

NOTE: Although mentioned elsewhere, it is useful to point out again that many inputs to the 1802, all of them being negative acting types, are pulled 'high' through resistors. This is done whether or not the ELF actually uses a particular input. The resistor pull-ups hold these negative acting inputs in their logic 0 state, but action by another circuit element can pull these inputs low (logic 1 state), In all such cases, these inputs are pulled low via diodes. Unused inputs that are normally pulled high can, in theory, be connected to unspecified circuits that might activate them. For example, if another 1-bit input is required, EF1, EF2, and EF3 are all available, and if an interrupt is required, the INT input (pin 36) is also available. The original ELF, as described in the Popular Electronics magazine article, mentioned these inputs, and showed them *being pulled high via resistors, but it did not go into any detail about WHY they were pulled high, what the resistors were for, etc.*

The LOAD toggle switch, SW2, is conditioned/debounced much the same way as with the IN pushbutton. When the switch is in its normal DOWN position, the switch common is connected to 0V and thus the junction of IC9 pins 6 & 5 will be = 1. This holds the flip-flop IC3 in its reset state via diode D3. Because the flip-flop is held reset, it cannot trigger the 1802's DMA IN input and the CPU will not perform a 'read the toggle switch data and store to memory' DMA operation. But when SW2 is in its UP position, IC9 is no longer able to hold the flip-flop in its reset state, and it is free to be clocked by the IN pushbutton SW1 and be reset by the CPU once it has committed to doing the 'switches to memory' DMA operation. *Note that the two diodes D3 & D4, along with pull-down resistor R9, act like another logic OR gate, but without needing to add another IC; when either of the diode anodes are activated, the common diode cathodes and top of R9 will = 1.*

Besides the above, when the LOAD switch is UP, IC9 pin 4 will = 1, and via diode D5 the 'N2/LOAD' signal to IC5 will = 1. *Note that as with D3/D4/R9 described above, the D5/D6/R10 combination acts like another logic OR gate, again without requiring the use of another IC.* The 'N2/LOAD' signal enables the logic that controls reading eight toggle switch's data and writing data to the display, and both of those things must be working during LOAD mode, which is why the LOAD switch SW2 controls the 'N2/LOAD' signal in this way. However, there is another instance, unrelated to LOAD mode, when it is necessary to enable reading the eight toggle switches and displaying data on the hex display……it is when those switches and displays are to be used as I/O devices during execution of a program. As described on page 27, whenever certain of the special instructions for I/O read/write (64, 65, 66, 67 or 6C, 6D, 6E, 6F) are executed, the 'N' register will have its third bit turned on (= 1), and this will result in the 1802's N2 output (pin 17) turning on (= 1). This will in turn pull the N2/LOAD line high via D6, enabling the toggle switch read and display write circuits.

NOTE: As hinted above, the ELF control circuits do not try to fully decode the possibilities of the 1802's N0, N1 and N2 outputs. Full decoding would result in seven discrete I/O control signals, instead of the one signal coming from N2 alone. This is why any of the I/O instructions that happen to have a binary 1 in their third most significant bit of their 'N' nibble will activate the ELF's toggle switch and display I/O circuits. To some degree, the same lack of full decoding is also apparent on the other 1802 control outputs such as SC0 & SC1, but the limited capabilities of the original ELF made full decoding unnecessary, and adding full decoding would have required more IC's, more wiring, and more cost to the hobbyist.

The CPU has an internal flip-flop that can be set and reset by special instructions. The 'Q' output of that internal flip-flop will = 1 when the flip-flop is set, and will = 0 when reset. The flip-flop's 'Q' output is in control of the 1802's 'Q' output (pin 4), so this pin will be high (5V) when 'Q' is set, and low (0V) when 'Q' is reset. 'Q' can be used in many ways, including operating simple on/off devices (lamps, relays, etc.), cycling slowly to flash LEDs, cycling faster to generate audio signals (the frequency of which depends on how fast 'Q' is cycled), and cycling very fast to generate serial digital signals. On the original ELF, 'Q' was only used for operating the on-board LED, although in theory 'Q' could also be connected externally to operate other devices.

As shown on the schematic, ‘Q’ also drives an audio output; a segment of IC14 is used as a buffer, and R/C network C5 & R13 act as a simple level shifter, converting the 0 ~ 5V digital signal to a +/- 2.5V signal that is better suited for connection to the input of an audio amplifier. Most audio amplifier inputs are nominally designed for a “line level” audio signal that might be +/- 1V or so, so the ELF’s +/- 2.5V signal might be on the ‘hot’ (loud) side. Most amplifiers will be able to tolerate this hot signal by turning their volume controls down. If not, the junction of C5 and R13 could be further conditioned by inserting a voltage divider, perhaps a series 68k resistor between the C5/R13 junction and the J2 jack, with a 33K resistor (from the junction of the 68k resistor and J2) to ground. This would form a 3:1 voltage divider, with an overall impedance of about 100k, between the C5/R13 level shifter and J2. With the order of magnitude increase in impedance between this and the upstream circuit, there should be negligible effect on the signal, other than dropping it to 1/3 voltage, or +/- 0.8V, more in line with the nominal “line level” signal. Note that J2 is set up as a stereo ‘phone’ type jack, which is the type commonly used for regular powered computer stereo speaker sets, so such a set could be directly plugged into J2 and used to listen to any audio tones that the ‘Q’ output may be programmed to generate.

As an alternate to having ‘Q’ drive the J2 audio output, IC14 pin 2 could be connected to a small relay’s coil, with the other side of the coil connected to +5V (it would need to be a 5V relay, although some 6V relays might work adequately). To avoid damaging IC14, the relay must have a low power coil, such as perhaps a reed relay. Discussion of further options in this regard is beyond the scope of this manual.

As described on page 27, the 1802’s MWR output (pin 35) is normally used directly to control the ELF’s RAM, by telling it when data on the Data Bus should be stored at the current memory address. This will happen when the MP switch SW3 is in its DOWN position, because MWR is connected through the switch directly to the IC1 & IC2 ‘R/W’ inputs. This is the ‘write enable’ signal and it is logically true when at 0V; if it is at 5V then the RAM cannot be written to. For this reason, SW3 will pull the ‘write enable’ signal to 5V when in its UP (Memory Protect) position, and the 1802’s MWR signal is not connected and thus ignored. Memory Protect must be off during program entry in LOAD mode, and it must be on when using LOAD to verify memory contents. Normally, Memory Protect can also be on during program execution, unless the program needs to modify the memory contents as part of its execution; such a program would malfunction if Memory Protect is turned on.

The three segments of IC5 are used to generate strobe signals for control of reading data from the eight toggle switches, and writing data to the hex display. One segment of IC5, pins 3 ~ 6, is a 3-input NAND gate, and with all three inputs connected together it simply acts like an inverter; it takes the 1802’s MRD output (pin 7) and inverts it so when the CPU is in a READ cycle (and MRD is low or 0V), IC5 pin 6 will = 1. MRD (active low) is used directly on the ELF because it connects to the RAM ‘OD’ (Output Disable) inputs, so when the CPU is trying to read data from memory, MRD will = 0, and thus ‘OD’ will also be low, meaning that the memory outputs are NOT being disabled, thus they are enabled, and can put their data onto the Data Bus for the CPU to read.

The hex displays (IC11 & IC12) only show data on the Data Bus, and only what was on that bus during the memory READ cycle. Accordingly, the MRD signal that has been inverted by IC5 as described in the above paragraph, will be = 1 during memory READ when memory data is on the Data Bus. This inverted signal is applied to another of the 3-input NAND gates in IC5, via pin 8. Another pin on the same gate (pin 1) will be = 1 when either the ELF is in LOAD mode, or when the 'N2/LOAD' signal = 1, meaning that the CPU is executing an I/O instruction and thus needs to access the toggle switches and/or displays. The third pin on the same gate (pin 2) is controlled by the TPB output of the 1802 (pin 33), which has to do with memory timing and is necessary in this instance. So when MRD is active AND the ELF is in LOAD (or the CPU is performing an I/O access operation) AND the 1802's TPB output is active, IC5 pin 9 will be = 0. This is a strobe signal to tell the display IC's (IC11 & IC12) that whatever data is currently on the Data Bus must be allowed into the display's internal memory and held there even after the strobe from IC5 pin 9 goes away. The displays will show whatever data was thus strobed into them. Note that display IC11 and display IC12 are TTL type devices, and should not be connected into a CMOS logic node along with other CMOS inputs; for this reason a spare buffer from IC6 is used between IC5 pin 9 and the strobe inputs of IC11 & IC12.

IC11 & IC12 are TIL311 devices, which are hybrids of TTL logic and LED display technology. Each of these ICs can latch in whatever 4-bit data is on their data inputs (pins 2, 3, 12, 13) whenever their negative acting strobe input (pin 5) is = 0. When the strobe input is no longer = 0, the data previously strobed in is held in the IC's internal memory and used by the display. Whatever 4-bit data is in the IC's memory is interpreted by the internal TTL logic as a pattern of LED dots, in order to form the desired dot patterns to display numbers 0 ~ 9 and letters A ~ F, which correspond to the hex values of the 4-bit data. IC11 is connected to the high order 4-bits of the Data Bus and IC12 is connected to the low order 4-bits of the bus, so they display the high order and low order hex digits of the data, respectively. As noted in the above paragraph, the TTL data inputs of these display IC's require buffering from the CMOS Data Bus, and eight buffer segments of IC 6 & IC7 are used for this purpose. These display IC's have the ability to blank their LEDs (turn them all off) regardless of what data is currently in the display memory, but on the ELF the associated BLANKING inputs (pin 8) are always pulled low (= 0) to disable this feature. These display IC's have integral constant current sources for the LEDs in the 0 ~ F dot patterns, so there is no need for current limiting resistors for these LEDs. These displays also have two 'decimal points' located both on the lower left and the lower right of the digits, and these particular points are not part of the TTL logic control and they are not part of the integral constant current source circuits either; they are simply brought out to pins 4 and 10, but since the decimal points are not used on the ELF these LED pins are not connected. Note that since IC11 & IC12 are each actually like two circuits (the TTL logic circuit, and the constant current source & LED array circuit) in one physical package, there are in fact two power supply pins, one for each of the internal circuits. Pin 14 is the 'Vcc', or power feed to the TTL circuit. Pin 1 is the 'Vled', or power feed to the constant current source for the display LEDs and also the two decimal point LED anodes. *If the decimal point LED's were to be used, they would need current limiting resistors.*

Continuing with what was started on the bottom of page 31, the third segment of IC5 (pins 10 ~ 13) is another 3-input NAND gate. It is activated when 'N2/LOAD' is = 1 and when the 1802's MRD output is =1, meaning that the CPU is not currently trying to read from memory, which implies that it MIGHT be currently be trying to do something else, including DMA reading from the eight toggle switches while in LOAD mode, or trying to read from I/O (the eight toggle switches), and reading from I/O is distinct from reading from memory. *This is an important distinction about the 1802 that can easily confuse programmers; most microprocessors that are contemporary to the 1802 used 'memory mapped I/O', in which case the CPU has no special instructions or dedicated pins for handling I/O, and all I/O devices are decoded as if they were simply part of regular memory, albeit in specific memory locations. The 1802 can still do memory mapped I/O, but in addition it has the aforementioned special I/O instructions and dedicated N0, N1, and N2 outputs to support that special I/O.* With the 'N2/LOAD' and MRD signals in the appropriate states as described above, the IC5 pin 10 output will pulse low when the Data Bus is available to accept data from the eight toggle switches. One inverter segment of IC9 (pins 3 & 2) changes this to a 'pulse high', which will activate the two banks of electronic switches, IC8 & IC10, which require a logic 1 to 'close' their internal solid-state switches.

IC8 & IC10 are an interesting type of IC, in that they are not actually logic devices, although they are CMOS devices in terms of their control inputs. Each of these IC's contains four electronic (solid-state) switch elements, numbered 1 ~ 4. Each switch is made from a pair of MOSFET transistors, and each can conduct in either direction, so they have no defined input or output. In this way, they behave almost identically to switch or relay contacts; when OFF they have an impedance that is very high, essentially infinity, and when ON they have an impedance that is usually only a few Ohms. They can handle digital or analog signals equally well, as long as the signals do not exceed the voltage supply to the IC. Taking IC8's switch '1' as an example, its solid-state 'contacts' are brought to 'X1A' (switch 1, side A) on pin 1, and 'X1B' (switch 1, side B) on pin 2. This switch is controlled by input 'C1' (control for switch 1) on pin 13. When 'C1' = 0, switch 1 is 'open' and there is a very high impedance, like an open circuit, between 'X1A' and 'X1B'. When 'C1' = 1, switch 1 is 'closed' and there is a very low impedance between 'X1A' and 'X1B'.

All 'C' inputs to both IC8 & IC10 are connected together and are driven by the 'read switches' pulse coming from IC9 pin 2. When the pulse occurs, all eight switches in these IC's will close, effectively connecting the eight toggle switches to the Data Bus. When there is no 'read switches' pulse, all eight switches in these ICs will open, effectively isolating the eight toggle switches from the Data Bus.

Note that on some datasheets for the 4016 part, the X1A, X1B, C1 style pin designations are replaced by X1, Y1, A1 style designations.

The eight toggle switches have their contacts connected to the 5V and ground power buses, so when any switch is UP the switch will present a 5V signal (logical 1) to the associated solid-state switch, and when the switch is DOWN it will present a 0V signal (logical 0). The 0's or 1's from all eight toggle switches are sent to the Data Bus at the same time during the 'read switches' pulse, which as stated previously occurs as part of the CPU's special I/O Read operations.

The 8-bit Address Bus on the ELF is used only for regular memory (remember that on the 1802 the N0, N1 & N2 outputs are essentially a 3-bit address bus for the special I/O operations). 8 bits of binary can only account for 256 possible states, and the ELF has only 256 bytes of RAM (memory), so there is no need in the ELF to add the more complex circuitry to handle the high order address multiplexing. The flip side to this is that the ELF is very limited to using only small programs, which is just as well considering the lack of non-volatile program storage and rudimentary toggle switch program input, and lack of an address display.

The ELF's RAM (memory) is entirely within IC1 & IC2, which are either 2101 type or 5101 type. Both types are functionally identical and have the same pinouts, with the only difference being that the 5101 is internally constructed using CMOS technology, which really makes them a better fit for the ELF circuits. Each 2101/5101 is a 256 x 4 'static RAM' device, where 'static' means that the memory does not need to be constantly read and re-written in order to remain viable (which is what needed to happen with 'dynamic' RAM devices). Back in the 1970's when the 1802 microprocessor came out, static RAM was very expensive, and so dynamic RAM was more common due to its lower chip price (but at the expense of greater complexity in the support circuits that handled the constant refreshing operations). At the price point and desired low complexity of the ELF, dynamic RAM was not feasible, dictating the use of static RAM, and the 2101/5101 was probably the lowest cost option given these requirements. The IC is called '256 x 4' because it contains 256 'nibbles' of memory, as opposed to the more common 'bytes' of memory, where a 'nibble' is only 4 bits wide. This is why two RAM IC's are required; one IC contains the high order nibbles and the other IC contains the low order nibbles. However, both IC's have their control inputs connected together, so they operate in tandem as virtual 8-bit memory devices.

The 8-bit Address Bus connect to both IC1 & IC2 via their address inputs (pins 1 ~ 7 & 21). The Data Bus connects to the IC's via their dual data lines for each bit; the 2101/5101 device uses separate pins for data input and data output, so on the ELF both data lines for a given bit are simply connected together. Pins 9 & 10 are used for the first bit, pins 11 & 12 for the next bit, and so on for pins 13 & 14 and 15 & 16 for the other two bits.

In a larger memory circuit, it would be necessary to have many banks of memory like the ELF's, each being a 'page' (256 bytes) in size. The low order 8 bits of the address would always connect to the address inputs of IC's like the 2101/5101, but it would be necessary to decode the high order 8 bits of the address and use the decoded signals to enable only one 'page' of memory at a given time. Because the ELF uses only a single 'page' of RAM, there is no need to decode the high order addresses. However, the 2101/5101 IC's still have provision for being enabled as small parts of a larger memory bank, and the 'CE1' & 'CE2' inputs are used for this purpose. Both inputs must be logically = 1 in order for the IC to 'wake up' and be responsive to the address inputs and other control inputs, and most specifically, to accept data that the CPU is trying to write into them, and to send out data that the CPU is trying to read from them. Since the ELF only has this one 'page' of RAM, these IC's can be enabled at all times, instead of being selected by memory decode logic. Accordingly, low-acting CE1 (pin 19) is connected directly to ground (logic 0) and high-acting CE2 (pin 17) is connected directly to V+/5V (logic 1).

The RAM IC's need to know whether the data on the Data Bus should be input as part of a CPU WRITE operation, or output as part of a CPU READ operation, and the 'R/W' inputs are used for this purpose. Recall that from page 31 the 'write enable' signal will be = 0 for writing to the RAM, and = 1 at all other times. The 2101/5101 'R/W' input (pin 20) is active high for READ and active low for WRITE, so when 'write enable' is active (= 0) the RAM IC's will regard this as being told to go into their WRITE mode, and when 'write enable' is inactive (= 1) the RAM

IC's will regard this as being told to go into their READ mode. Obviously, the data input pins are disregarded unless the RAM IC is in WRITE, and the data output pins are inactive unless the RAM IC is in READ (recall that WRITE and READ are from the CPU's perspective).
Besides being in READ mode, another control input is required to activate the RAM IC's data output pins, and this is the 'OD' (output disable) on pin 18. 'OD' is driven by the CPU via its 'MRD' output (pin 7), and recall from above that 'MRD' is active (0V since it is negative acting) only during a CPU READ cycle, when data is to be read from memory. The RAM's 'OD' inputs will disable the ICs' data outputs when = 1, and enable those data outputs when = 0, so the logic between the 'MRD' signal and the 'OD' inputs is correct.

This completes the circuit description of the ELF.

MORE ABOUT THE ELF'S DMA

Although already covered in the circuit description chapter, the ELF's DMA operation is so important and unique that it will be treated with a brief overview here.

DMA is the mnemonic for Direct Memory Access. The 1802 is equipped with special pins dedicated to DMA operations, and it has unique internal functionality to support DMA.

Activating the DMA IN input (pin 38) will cause the CPU to immediately read a byte of data from an input device and then store it in a memory location without intervention by the program being executed. On the ELF, the primary input device for DMA is the bank of eight toggle switches.

Activating the DMA OUT input (pin 37) will cause the CPU to immediately transfer a byte of data from a memory location to an output device. On the ELF, there is an output device in the form of the 2-digit hex display, but it is connected logically in such a way that it operates normally as part of the DMA IN operation instead of as part of a DMA OUT operation. Accordingly, the DMA OUT input is not used on the ELF; however, it is pulled high by resistor R3 and could be used in theory.

An integral memory pointer in the CPU is used to indicate the memory location to be used for the DMA operations. For normal (non-LOAD mode) operations, the program will initially set this memory pointer to the desired starting memory location for DMA operations. After each DMA operation, the memory pointer is incremented to point to the next memory location. So, for example, if an external device needed to load a bunch of data into the ELF's memory starting at address 0C00, the program would initialize the memory pointer to 0C00, and then go about its business. The external device would then make the first byte of data available to the ELF, and the device would toggle the DMA IN input by bringing pin 38 low and then high again. The CPU would respond by reading the data byte from the external device, storing it in memory location 0C00, and incrementing the memory pointer to 0C01. The external device could continue presenting data bytes and toggling the DMA IN, and eventually all data would be loaded into memory, at which point the external device would stop toggling the DMA IN. During this whole DMA operation, the CPU would be continuing to run whatever program it was busy with, and the DMA operations would be fit into the CPU's cycles as required. If an external device needed to read a bunch of data from memory, a similar method would be used, but the external device would toggle the DMA OUT input instead of the DMA IN input.

The above is also basically what happens when the ELF is in its LOAD mode. The LOAD switch SW2 is put into its UP position, which releases the constant reset of flip-flop IC3 and pulls the CPU's WAIT input low, forcing any running program to stop executing. If the Run switch SW4 was previously in its DOWN position, the CPU would have been cleared, including establishing register R0 as the program counter (memory pointer for DMA operations) and resetting R0 to the first memory location 0000. Now, with the CPU cleared and reset, a DMA memory pointer established, any running program stopped, and the flip-flop operation enabled by releasing its reset, repeated pressing of the IN pushbutton SW1 will toggle the DMA IN input (via the flip-flop), reading in data from the eight toggle switches, and storing to sequential memory locations.

USING THE ELF

The ELF basically has four modes of operation, as listed in the following table. The 'LOAD' and 'RUN' columns refer to the LOAD and RUN toggle switches SW2 & SW4, and a '1' in their columns means that the associated switch is in its UP (on) position, whereas a '0' means the DOWN (off) position.

LOAD	RUN	MODE
0	0	CLEAR / RESET
1	0	LOAD / REVIEW
0	1	RUN PROGRAM
1	1	WAIT / PAUSE

The 'MP' toggle switch must be in its DOWN (off) position when loading a program from the toggle switches, and it must be in its UP (on) position when reviewing a program. It is the setting of the 'MP' switch that determines whether the ELF is in LOAD or REVIEW mode when the LOAD switch is UP and the RUN switch is DOWN.

Basic Operation

1) Set LOAD , RUN and MP to their DOWN positions to reset the CPU and program counter to 0000 (hex) and enable the RAM for receiving data from the toggle switches
2) Raise LOAD to its UP position, putting the ELF into LOAD mode
3) Set the first byte of the program in binary on the eight toggle switches
4) Press and release the IN pushbutton SW1
5) The program data that was just entered into memory is displayed in hex
6) The CPU automatically increments the program counter to the next memory address

Note that the memory address/location is NOT displayed anywhere on the ELF. You must keep track of the address manually.

7) Repeat steps 3 ~ 6 for every other byte of the program
8) When the program has been entered, lower LOAD to its DOWN position. Since RUN is still DOWN, the two switches both being DOWN causes the CPU to reset its program counter to 0000 (hex)
9) Raise MP to its UP position, preventing the ELF from changing memory contents
10) Raise LOAD to its UP position. The ELF is now in REVIEW mode since MP is UP
11) Repeatedly press IN to sequence through memory; each byte of the program will be displayed in turn, so you can verify that all was entered correctly - *see special instructions below if you find an incorrectly entered byte of data*
12) Once the program was been verified, lower LOAD to its DOWN position.
13) If the program you entered will need to modify memory contents as part of its execution, lower MP to its DOWN position, otherwise leave it in its UP position
14) Raise RUN to its UP position and the ELF will be running the program you entered

15) During RUN, if you wish to temporarily pause the program execution without losing any status, raise LOAD to its UP position, and lower it to its DOWN position to resume program execution; this is WAIT/PAUSE, and the CPU is still technically running while paused, although nothing is being done
16) To stop running the program, lower RUN to its DOWN position. To run it again from the start, raise RUN to its UP position

Fixing Incorrect Program Data

If a mistake is made when entering program data, follow this procedure to fix the error.

1) If the error is in the first byte of the program (address 0000 hex), set the ELF up as you would for normal program entry, with MP in the DOWN position, set up the eight toggle switches with the correct data, and press IN once, then raise MP to its UP position.
2) If the error was in a byte other than the first byte (not in address 0000 hex), set up the ELF to REVIEW the program (MP will be UP) and press IN repeatedly until the byte <u>immediately before</u> the erroneous byte is displayed, set the correct byte onto the eight toggle switches, lower MP to DOWN, press IN once, return MP to UP. Repeat if there are any other errors to correct.
3) For either of the above situations, after correcting the error as described, set the ELF for normal program REVIEW and sequence through the program, at least in the area immediately before, during and immediately after the location of the fixed byte, to make sure that all program data in that area is intact and you have really corrected the error.

SAMPLE PROGRAMS

Upon completing the construction of a new ELF microcomputer, the builder will want to immediately test it to verify that it works. A brief program for testing the ELF is listed below, along with three other programs that between them demonstrate most aspects of ELF operation, and in the course of doing so they illustrate many aspects of how the 1802 instruction set works.

Lamp Flasher

This is a version of a classic program used to test the ELF. It does one thing only; it flashes the 'Q' LED off and on repeatedly until the program is stopped.

Address	Opcode	Mnemonic	Description
0000	7A	REQ	Reset 'Q' = 0 (turns 'Q' LED off)
0001	F8	LDI	Load data (10) into 'D' register
0002	10	(data)	*data for preceding instruction*
0003	B1	PHI	Set high order byte of register R1 = 'D' (10)
0004	21	DEC	Decrement contents of register R1
0005	91	GHI	Set 'D' equal to contents of high order byte of R1
0006	3A	BNZ	If 'D' ≠ 0, short branch to address 00***04***
0007	04	(data)	*data (low order address) for preceding instruction*
0008	31	BQ	If 'Q' = 1, short branch to address 00***00***
0009	00	(data)	*data (low order address) for preceding instruction*
000A	7B	SEQ	Set 'Q' = 1 (turns 'Q' LED on)
000B	30	BR	Unconditional short branch to address 00***01***
000C	01	(data)	*data (low order address) for preceding instruction*

The program first turns the 'Q' LED off by executing the REQ instruction, resetting the CPU's internal flip-flop, so it's 'Q' output will be = 0. Then the program establishes and runs a time delay loop; a value (10) is loaded into the high order byte of register R1 (remember that unless a program changes it, register R0 is the default program counter that keeps track of where the CPU is in the running program, so we can't use R0 for this time delay), and the low order byte of R1 is left unmodified (it is probably 00 but it does not matter much in this instance), so R1 is now set to 1000 hex. The time delay loop proceeds by decrementing R1, which means it goes from 1000 to 0FFF, then to 0FFE, then 0FFD, etc; every time it gets decremented. After every decrement, the program checks to see what value R1 contains, and it does this by copying the contents of the high order byte of R1 to register 'D' (the 1802 does not have an instruction for testing the contents of any 16-bit register directly, so all such operations must be accomplished by moving the data into the 'D' register, which DOES have instructions for testing its value). The time delay loop then performs the test by using the BNZ (branch on 'D' not zero), so if the value in 'D' is NOT equal to zero (i.e. R1 has not yet been decremented down to at least 00??), the programmed time delay loop does a short branch back to address 0004, which continues the decrement process of R1. But if R1 has in fact been decremented down to the point where its high order byte is now 00 hex (again, we don't much care if the low order byte is zero or not, because at the high rate of speed the CPU is running, such insignificant low order parts of the register value have negligible impact on the time delay), then the time delay is complete and the program can move on to address 0008, where it executes the BQ instruction, which checks the current value of the integral flip-flop (i.e. it checks ‘Q’) to see if it is = 1. If ‘Q’ is indeed = 1, then the program does a short branch to address 0000, where it turns ‘Q’ back to = 0, so the ‘Q’ LED will be off. If ‘Q’ is ≠ 1, the program advances to address 000A, where the SEQ is executed, setting the flip-flop so its ‘Q’ is = 1 and the ‘Q’ LED will be on. Then the program advances to address 000B where the BR instruction is an unconditional (it always works and does not rely on some other thing being true or false in order to operate) short branch back to address 0001, the start of the time delay loop. It does not branch back to 0000 because that would only turn ‘Q’ off, and the program just finished turning ‘Q’ on in the previous instruction. So the program toggles ‘Q’ on and off (starting with ‘Q’ off) and does a time delay loop in between every toggle operation, so the flashing is not too fast to see. It turns out that the 1802 operating with a 1MHz clock will take roughly half a second to decrement a register such as R1 from 10?? down to 00?? (where ?? simply means that we did not specify the value of the low order byte and don’t care what it is). To make this program flash the ‘Q’ LED faster, a value less than 10 hex should be put in address 0002, and to flash the ‘Q’ LED slower, a value greater than 10 hex should be put in address 0002.

Pushbutton Interrogator

This is a classic program from the original Popular Electronics magazine article in which the ELF was announced. It watches the IN pushbutton and turns on the ‘Q’ LED when IN is pressed. It demonstrates how the ELF allows the status of the IN pushbutton to read by a program and acted upon (as opposed to the IN pushbutton being used only for entering a program).

Address	**Opcode**	**Mnemonic**	**Description**
0000	7A	REQ	Reset ‘Q’ = 0 (turns ‘Q’ LED off)
0001	3F	BN4	If ‘EF4’ = 0, short branch to address 00***00***
0002	00	(data)	*data for preceding instruction*
0003	7B	SEQ	Set ‘Q’ = 1 (turns ‘Q’ LED on)
0004	30	BR	Unconditional short branch to address 00***01***
0005	01	(data)	*data (low order address) for preceding instruction*

The program first turns the ‘Q’ LED off by executing the REQ instruction, resetting the CPU’s internal flip-flop, so its ‘Q’ output will be = 0 (turning the ‘Q’ LED off). Then the program executes the BN4 instruction, which is a conditional short branch that is allowed to work only if the 1802’s EF4 input is currently at a logic 0 (but it is active low, so the BN4 instruction is actually checking to see if the EF4 input is at 0V, which is what happens if you press the IN pushbutton). So if the IN pushbutton is NOT being pressed, EF4 will = 0 and the program constantly loops between addresses 0000 and 0002, repeatedly asserting that ‘Q’ should remain reset (‘Q’ LED off). But when the IN pushbutton is pressed, EF4 will = 1 and the BN4 instruction will not work as a result, and program execution will continue with the next instruction at address 0003. This is the SEQ instruction, which sets the CPU’s internal flip-flop, so its ‘Q’ output will be = 1 (turning the ‘Q’ LED on). Then the program advances to address 0004 where the BR instruction is an unconditional (it always works and does not rely on some other thing being true or false in order to operate) short branch back to address 0001, where EF4 is tested again, and assuming that the IN pushbutton is still pressed, the BN4 will fail again and the program will again assert that ‘Q’ should = 1, and program execution continues to work as described above. So the program simply loops rapidly and sets or resets ‘Q’ according to whether the IN pushbutton is currently pressed or not.

Pushbutton Counter

This is a classic program from the original Popular Electronics magazine article in which the ELF was announced. It waits until you have pressed the IN pushbutton a specified number of times, and then turns on the ‘Q’ LED. It demonstrates how the ELF allows the status of the IN pushbutton to read by a program and acted upon (as opposed to the IN pushbutton being used only for entering a program). It also demonstrates a method for implementing a counter.

Address	**Opcode**	**Mnemonic**	**Description**
0000	F8	LDI	Load data (05) into ‘D’ register
0001	05	(data)	*data for preceding instruction*
0002	3F	BN4	If ‘EF4’ = 0, short branch to address 00***02***
0003	02	(data)	*data (low order address) for preceding instruction*
0004	37	B4	If ‘EF4’ = 1, short branch to address 00***04***
0005	04	(data)	*data (low order address) for preceding instruction*
0006	FF	SMI	‘D’ = ‘D’ - immediate value in next byte (subtract)
0007	01	(data)	*data for preceding instruction*
0008	3A	BNZ	If ‘D’ ≠ 0, short branch to address 00***02***
0009	02	(data)	*data (low order address) for preceding instruction*
000A	7B	SEQ	Set ‘Q’ = 1 (turns ‘Q’ LED on)
000B	30	BR	Unconditional short branch to address 00***0A***
000C	0A	(data)	*data (low order address) for preceding instruction*

The program first loads the immediate value of 05 into the ‘D’ register. Then the program uses the BN4 instruction to test whether the EF4 input is at logical 0 (the negative acting EF4 input is at 5V), which means that the IN pushbutton is not being pressed; if IN is not being pressed, then BN4 does a short branch back to address 0002, which is its own address, thus it repeats its operation. As long as IN remains un-pressed, the program will constantly loop here, repeating the BN4 instruction and testing the EF4 input to see if the status of the IN pushbutton has changed. Once IN has been pressed, EF4 will finally be at logical 1, and the BN4 instruction will fail, so program execution continues at address 0004, where the B4 instruction tests to see if EF4 = 1. Since IN is still pressed, EF4 will = 1 and B4 does a short branch back to address 0004, which is its own address, thus it repeats its operation. As long as IN remains pressed, the program will constantly loop here, repeating the B4 instruction and testing the EF4 input to see if the status of the IN pushbutton has changed. Once IN has been released, EF4 will finally be at logical 0 again, and the B4 instruction will fail, so program execution continues at address 0006, where the SMI instruction decrements the value in ‘D’. Since the ‘D’ register does not have a decrement instruction, the decrement is done by subtraction of the following value of 01, with the thus decremented result remaining in ‘D’. After every decrement of ‘D’, the program uses the BNZ (short branch if ‘D’ is not equal to zero) to check whether the value in ‘D’ has been decremented all the way down to 0, or not. Assuming that ‘D’ is not yet = 0, the BNZ does its short branch back to address 0002, where the whole process of testing the IN pushbutton and decrementing ‘D’ repeats. In this way, every press-and-release cycle of the IN pushbutton will result in ‘D’ being decremented. When ‘D’ has finally decremented down to = 0, BNZ fails and the program execution continues at address 000A, where the SEQ instruction sets the CPU’s internal flip-flop, so its ‘Q’ output turns on, which causes the ‘Q’ output to turn on, and the ‘Q’ LED turns on. Program execution continues at address 000B, where the BR instruction is an unconditional (it always works and does not rely on some other thing being true or false in order to operate) short branch back to address 000A, where the program reasserts that ‘Q’ should remain set, and this the ‘Q’ LED turned on. The program is now stuck in this final loop and will not exit the loop until the ELF is taken out of RUN mode. So the program starts with a preset of 05 hex (5 decimal) and after the IN pushbutton has been pressed that many times, it turns the ‘Q’ LED on and waits indefinitely for you stop program execution. To change the number of times that the IN pushbutton needs to be pressed before the ‘Q” LED turns on, change the data in address 0001.

Using the I/O Instructions

This is a program from Lee Hart’s “1802 Membership Card” (a variation on the ELF) website, http://www.retrotechnology.com/memship/mship_test.html. It demonstrates how the 1802 can be programmed to read from, and write to, I/O using its special I/O instructions. In this way, the ELF’s eight toggle switches can be read as input data by a running program, and a program can write data as output to the ELF’s 2-digit display; the switches and display can thusly be used for something other than program entry.

Address	Opcode	Mnemonic	Description
0000	E1	SEX	Load instruction ‘N’ value (1) into ‘X’ register
0001	90	GHI	Set ‘D’ equal to contents of high order byte of R0
0002	B1	PHI	Set high order byte of register R1 = ‘D’
0003	F8	LDI	Load data (0A) into ‘D’ register
0004	0A	(data)	*data for preceding instruction*
0005	A1	PLO	Set low order byte of register R1 = ‘D’
0006	6C	INP4	Set Memory (R(X)) = data from I/O (Input Switches)
0007	64	OUT4	Output data from Memory (R(X)) to I/O (Display)
0008	30	BR	Unconditional short branch to address 00***00***
0009	00	(data)	*data (low order address) for preceding instruction*
000A	xx	(data)	*temporary data storage between switches & display*

NOTE: Contrary to the basic ELF operating instructions, the MP switch SW3 must be in the DOWN position for this program to work correctly, since it needs to write into location 000A.

The program first sets the 4-bit ‘X’ register = hex 1 (decimal 1). The current program counter is R0, and the program next uses instruction 90 (Get High Register N) to copy the high order byte of R0 to the ‘D’ register; R0 is specified by the ‘N’ part of the instruction opcode 9<u>0</u>. Then instruction B1 (Put High Register N) to set the high order byte of register R1 = the value in ‘D’, which was previously the value of the high order byte of R0; R1 is specified by the ‘N’ part of the instruction opcode B<u>1</u>. Then the program executes the F8 instruction (Load ‘D’ Immediate) at address 0003 to set ‘D’ to the data in the next location (0A). Next the A1 instruction (Out Low Register N) to copy the data from ‘D’ to the low order byte of register R1; R1 is specified by the ‘N’ part of the instruction opcode A<u>1</u>.

So far, the high byte of program counter R0 is copied to the high byte of R1, and the low byte of R1 is set to a value of 0A, so R1 contains 000A, which is the address of the temporary data memory location at the end of the program.

NOTE: An interesting programming technique is demonstrated here. Since we know the program as shown resides in the lowest ‘page’ of memory (with the high order byte of the address being 00 hex), we could simply set the high order byte of R1 using LDI and PHI, similar to what is done to set the low order byte of R1. But the program’s author wrote it to be tolerant of which ‘page’ the program resides in, and thus checks the current program counter R0’s high order byte to see which ‘page’ the program is actually in, and uses this value to set the high byte of R1, so that it points to the correct address.

Then the program executes the 6C instruction (Input); this one is a bit tricky to understand. First, the ‘N’ portion of the instruction (C) is placed in the CPU’s 4-bit ‘N’ register, so that register contains a value of C hex. As described on page 27, the binary value from the least significant three bits of the ‘N’ register are used to control the three I/O control outputs N0, N1 & N2. The most significant bit of ‘N’ is used to specify whether the I/O controlled by those three outputs will be Input or Output types. C hex is 1100 in binary, so the most significant binary digit of 1 means that this is an Input fetch from I/O, and the binary 100 in the lower three bits has a value of 4, so Input I/O device 4 will be accessed. Note that the outputs N2, N1, N0 correspond to the bits 100 in this example, from left to right, so N2 = 1, N1 = 0, and N0 = 0. On the ELF, output N2 is used to control the strobes to the eight toggle switches and the 2-digit hex display, so instruction 6C, by virtue of its ‘C’ will input data from the eight toggle switches. As part of the execution of the 6C instruction, the switch data is stored in the memory location addressed by the contents of the scratch-pad register specified by the value in the ‘X’ register (1). So, the CPU looks in register R<u>1</u>, and finds the value 000A, and uses it to address that memory location, which we know is the designated temporary storage location for switch data, and the switch data is therefore stored at location 000A.

Then the program immediately executes the 64 instruction (OUTPUT); this one works much like the previous 6C instruction, but it is working to copy data from a memory location and send it to an output device, which on the ELF is the hex display. As with instruction 6C, the ‘N’ portion of the instruction (4) is placed in register ‘N’ and treated like binary data, or 0100. Since the most significant bit in ‘N’ is a 0, this tells the CPU that data will be Output to an I/O device. The least significant three bits in ‘N’ control the outputs N0, N1 & N2, and note that the 3-bit binary pattern here is the same as with the 6C instruction, so N2 will =1, N1 = 0, N0 = 0, and the fact that the N2 output = 1 will control the ELF logic such that the Output data will go to the 2-digit hex display. This 64 instruction also uses the memory location addressed by the value in the register specified by the value of the 'X' register, in order to know which memory location contains the data that will be output to the display; 'X' still contains a 1, so register R1's contents, which are still 000A, are used as the address of the memory location containing the data to be output. Since this memory location already contains the data that was read in from the eight toggle switches, that same data will now be output to the display. After all that is completed, the 64 instruction then increments the data in register R1, so it will now contain 000B; *for our purposes in this program, we don't need it to do this, but it does…..see below.*

Finally the program executes instruction 30 (an unconditional short branch) at address 0008, and this uses the data 00 from the next memory location to branch to memory location 00<u>00</u>, the start of the program. So, the program continuously runs as described above, setting up the 'X' register as 1 and the R1 register as 000A, reading the switch data, saving the data to address location 000A, writing that data from 000A to the display, and repeating indefinitely. Thus the display will always instantly show the hex value of the binary data from the eight toggle switches.

The program starts with, and constantly repeats, the setting of register R1 = 000A. This is necessary because the OUT instruction at address 0007 has the undesired effect of incrementing R1 every time it executes. If it were not for that, the 'BR' instruction at address 0008 could be made to branch back to address 0006 instead of branching back to the start of the program as it does.

BIBLIOGRAPHY

The following documents were used as a basis for the author’s ELF design/build and the contents of this manual:

1) Article “Build the COSMAC “ELF” – A Low Cost Experimenter’s Microcomputer” by Joseph Weisbecker, from the August 1976 issue of Popular Electronics magazine
2) Schematic diagram of the commercial product ‘Quest COSMAC ELF’
3) Information from Lee Hart’s website on the 1802 Membership Card (ELF inspired computer), www.retrotechnology.com/memship
4) Schematic diagram of the 1802 Membership Card microcomputer kit
5) Articles and FAQ’s from the http://cosmacelf.com website
6) “User Manual for the CDP1802 COSMAC Microprocessor”, RCA publication MPM-201A *(which contains errors, in its lists and descriptions of certain instructions, that have been corrected herein)*
7) “The 1802 Instruction Set”, article by Dann McCreary, from the Compute II, Issue 3, August/September 1980, Page 52
8) “CMOS Cookbook” by Don Lancaster, Howard Sams # 21398
9) Texas Instruments datasheets for the 4013, 4016, 4023, 4049, 4050, 4584/40106 CMOS logic IC’s
10) NEC datasheet for the uPD5101L 256 x 4 static RAM IC
11) Texas Instruments datasheet for the TIL311 integrated hybrid display
12) ECS datasheet for the ECS-100AC-010 hybrid oscillator
13) “Programs to Read the Data Switches, and Display Them on the LEDs”, by Lee Hart, Feb 21st 2010, shorter version & read data switches Sept 5 2010 – *program on page 42 of this manual used by permission*

ABSTRACT & DISCLAIMER

The author read the original Popular Electronics magazine article (August 1976) for the ELF microcomputer when it was first published, but was unable to afford one at that time (having previously lusted after the Altair 8800 microcomputer). Eventually the author had a career in electronics, industrial design, music, and finally decided to either reacquire or build examples of all early microcomputers he had used in his life, and use them for demonstrations at schools and other events, and document them in a series of videos on YouTube, and write about them. The ELF build was one of the last computers to be acquired as part of this project.

The author was disappointed in some of the documentation available, and also wished to make it easier for others to read about, study, learn from, and possibly build, the original *unexpanded* ELF computer. To this end, the author's well-researched ELF build was fully documented, and the sum total of the documentation has been carefully rewritten as this set of documents (user's manual included). The intent is to provide in one place all information needed to build an example of the original ELF, understand how it works, and how to program and use it.

COLOPHON

- The schematic diagram and parts layout/dimension drawings were created using AutoCAD 2010, and rendered to PDF using Adobe Acrobat
- The parts/supplier list was generated using Microsoft Excel 2010, and rendered to PDF using Adobe Acrobat
- The user's manual was generated using Microsoft Word 2010, and rendered to PDF using Adobe Acrobat
- The toggle switch panel design was created using Front Panel Designer 5.01, by Schaeffer AG, Germany, and rendered to PDF using Adobe Acrobat
- The IC inverted pin labels were created using Front Panel Designer 5.01, by Schaeffer AG, Germany, and rendered to PDF using Adobe Acrobat
- Photos were edited using Serif PhotoPlus X7