M. P. Publishing Co. Box 378 Belmont, Mass. 02178 Vol. 1 No. 4 April '75

ECS

THE MONTHLY MAGAZINE OF IDEAS
FOR THE MICROCOMPUTER EXPERIMENTER

Publisher's Introduction:

Here you have the April 1975 issue of ECS, complete and unexpurgated. The main theme of this issue is the introduction of the "SIRIUS-MP" language as a notational form for expressing programs. The idea of SIRIUS-MP is to slightly generalize the low level code approach to program notation so that it will be fairly expedient for subscribers to hand "cross compile" programs on whatever variation of the "home brew computer" concept they have implemented. The variations on this theme include...

1. The SIRIUS-MP Language... This article, beginning on page 2, is a first statement in these pages of some of the concepts involved in the language. It also provides information useful in understanding the several SIRIUS examples found in this issue.

2. BOOTER: An "Emergency" Bootstrap Loader... It is common knowledge what to "do when the lights go out." But what do you do after the lights go out when your computer and volatile software were on the same power source as the lights? Turn to page 11 for a description of an emergency bootstrap loader concocted one weekend to combat electron deficiency anemia.

3. IMP Extensions For Tape Interface Control (Continued...) In the last issue, I did not quite fit all I intended to print within the confines of 28 pages. The remaining segments of the tape interface are presented in a SIRIUS fashion along with the equivalent 8008 code, beginning on page 14.

4. Comments on the ECS-8 Design: Turn to page 19 for a short note on one aspect of the ECS-8 design which I should have pointed out in the March article, and was the source of a complaint from my brother Peter Helmers.

5. Notes on NAVIGATION IN THE VICINITY OF α-AQUILA ... #1. So, you went out and got yourself an Altair computer? Now what? Turn to page 20 for the first in a continuing series of articles on the use and abuse of the Intel 8080 instruction set in an ECS context - with occasional intermingled information on hardware interfaces to be supplied from time to time (but not this time however.)

6. Erratum: Turn to page 24 for a short note about an ECS-7 diagram error.

7. A Note Concerning The Motorola 6800 MPU: Also on page 24 is a short note concerning the use of the M6800 in an ECS context, now possible to contemplate on a practical basis in the near future.

This issue is going to press April 21 1975. The next issue is fairly well defined as of this date, and will include: an article by subscriber James Hogenson concerning the design of a unique oscilliscope graphics interface featuring a 4096 point (64 x 64 grid) matrix of spot locations; a continuation of the software discussions begun in this issue; and possibly a review of one or two tools which will be of interest to readers.

Carl T. Helmers, Jr.
Carl T. Helmers, Jr.
Publisher April 20 1975

The SIRIUS-MP Language...

an approach to machine independent low level code.

This issue begins a subject which will continue in the pages of ECS for some time to come: the subject of expressing programs in a fairly well defined low level "language" which is in principle independent of any particular microprocessor or other small computer you might have. This will facilitate your use of published programs written for an 8080 if you own an IMP-16, or programs written for 8008 if you own an M6800, etc. - provided the programs in question are expressed in the SIRIUS way.

The name I have chosen for this language is "SIRIUS-MP". The SIRIUS is a combination of an April pun and the following input: if Altair is the brightest star (visual magnitude) in the constellation Aquila, then let me modestly name this mode of program expression after the brightest star in the sky, the star α-Canis Major or SIRIUS. So, if you are SIRIUS about Altair (or other computers available inexpensively both now and in the near future) you will find this series of articles illuminating. So much for the advertisement now to turn to some information content....

WHAT IS A COMPUTER LANGUAGE?

The answer to this question (as is always the case with complicated subjects) can range from the superficial to the formal mathematical intricacies of compiler-writing and language design. Since this publication is not a technical journal on software engineering, it must necessarily leave out a lot of the detailed information on the subject, to concentrate on the application of the concept. (Upon sufficient interest - one inquiry- I'll spend an evening sometime and compile a bibleography on the subject of compilers and computer languages.) With this disclaimer I'll proceed to the subject of computer languages in the context of a home brew microcomputer system.

Starting from first principles, what is a "language" (eg: English, German, Pidgin, integral calculus, set theory) in general? I'll confine the subject arbitrarily to the concept of "written languages" and put forth the following formulation:

A LANGUAGE IS A HUMAN INVENTION FOR THE PURPOSE OF EXPRESSING THOUGHTS.

This definition is filled with implications: language is an invented technology (probably the first) of humans (or other critters.) language is utilized in communicating thoughts between individuals. Language is appropriate to thinking beings. Now what could this possibly have to do with your urge to program and use a microcomputer ?

A fair amount of course! The specific application of the language concept to the problem of programming a computer is the concept of a "programming language." The specific part of this application is the limiting of computer languages to certain classes of thoughts...

A COMPUTER LANGUAGE IS A HUMAN INVENTION FOR THE PURPOSE OF EXPRESSING COMPUTER PROGRAMS.

Just as there are numerous variations on the "natural language" concept (Eg: ENGLISH), the diversity of human thought has lead to a wide range of computer languages from the most general to the specific and application oriented. In each such language, the author(s) have selected a set of elements needed to solve the particular problem and combined these in a (more or less) self consistent manner and come up with a solution to the problem of expressing programs of a particular class.

The creation of a programming language for the particular case of a microprocessor system in the "homebrew" (ie: limited hardware) environment is the object of this series of articles in ECS. When you design and or build a hardware system, your first problem is solved - a computer that "works". To get beyond this first phase the problem becomes developing the programs enabling your system to do interesting things. A language can be used for ≥3 purposes in the process of programming your computer:

a. An appropriate language enables you to abstractly specify a program in a first iteration of design without worrying "too much" about details. Get the control flow figured out first, then worry about low level subroutines!

b. An appropriate language will enable you to hand compile programs expressed in that language for use on your own computer, even if the program was developed and debugged on another computer. You know the "algorithm" works even though you have not yet translated it to your own use.

c. A language appropriate for the home microprocessor will be of sufficient simplicity to allow hand compilation or compilation by a very simple compiler.

These considerations - the definition of a "home brew computer" context - are a major input into the design of the SIRIUS-MP method of program expression.

SETTING THE PROJECT IN CONTEXT: HOW WILL SIRIUS-MP COMPARE TO EXISTING LANGUAGES?

The approach taken in the choice of elements for the SIRIUS-MP language is that of a "pseudo assembly language." An assembler is the simplest of all software developme aids to write, so this choice tends to satisfy criterion "c" above. But what about "a" and "b"? This is where the "pseudo" part enters the description: it is a language one step removed from the detailed instruction level in many of its operations. SIRIUS is an assembly-type language for a class of similar machine architectures - with operations found in general on such machines forming its "primitives." The subject of addres resolution is left intentionally non-specific and symbolic so that variations in the way

data is accessed can be left to the hand or machine-aided process of generating code for your own system. Many of the statements written in this form will generate only a single instruction on the "object" machine - but others will require a series of several instructions to specify required actions on a given machine. It is my intention to include within this "pseudo assembly language" concept several programming constructs borrowed from high order languages in current usage - but stripped of the complex syntax of a true high level language and specified in the simplified form of the SIRIUS-MP syntax, such as it is. This adaptation of a language to a specific purpose and class of users is a widespread practice in the compiler/language design business. Several examples come to mind of specific languages for specific usage contexts:

XPL - this language is the compiler-writer's language to a great extent. It is a specific and limited subset of PL/1 by McKeeman, Wortman and Horning which isdocumented in a book entitled "A Compiler Generator." The adaptation here is to concentrate on those features necessary for the writing of compilers and exclude all else. (Intel PL/M is very close to XPL)

HAL/S - this language was developed for guidance, navigation and control applications of NASA by Intermetrics Inc., the author's employer of several years. HAL/S is specialized to include the vector and matrix data forms used in spacecraft navigation - and to provide highly visible "self-documented" code which was not possible in the assembly language style approach used in the Apollo program.

SNOBOL - here is a language which is primarily oriented to "string handling" programs - a very broad range of applications, in some sense including the writing of compilers as well.

ALGOL - this language is the antecedent of many currently used languages, whose original intent was a specialization in generality - the ways in which algorithms could be best specified, in the abstract form.

These languages are all examples of much more extensive and complex methods of program expression from a compiler writer's standpoint - although from the user's standpoint they are orders of magnitude easier to program with than doing the equivalent in a low level "pseudo assembly language" or formal assembly language for a specific machine. It is the problem of generating code by hand or with minimal program aids which limits the possibilities of SIRIUS program specifications to the low level approach.

WHAT ARE THE COMPONENTS OF A COMPUTER LANGUAGE?

For those readers with a software or computer-science background, this discussion is in the nature of a review. For readers with little programming background this will present new information.

When you build a computer from a kit or from scratch, your problem is to put together a set of hardware components according to a certain system design (usually inherent in the microcomputer chip design) such that all the components play together as a working system. At a level of abstraction far removed from - yet still within the context of - the detailed hardware, a language for computers is also a construction of component parts which must "play together" according to a particular design if the language is to be

useful as a means of expressing programs. At the most abstract level of discussion, a language consists of two major component parts designed to provide an interface between a human being's thoughts and the requirements of computing automata. These are:

SYNTAX: - this component of the language is the set of rules concerning the correct formulation of basic "statements" or "expressions" in the language in question.

SEMANTICS: - this component of the language is the set of rules governing the intelligible combinations of syntax elements - the combinations which produce a well defined and translatable meaning which can be used in turn to generate machine code for some "object" or "target" machine of a compiler.

The syntax and semantics of a programming language can be chosen with a somewhat ill-defined border: one of the major trade-offs to be done in designing a language and associated compiler is deciding how much of the work is to be performed by the syntactical analysis and how much is to be left to semantic interpretation. At one extreme there is the complex syntax of a high order language in which much of the semantic intent of a statement is inherent in the syntax used; at the other extreme there is the case of the simple "assembly language" style of syntax in which very little function is inherent in the syntax - which merely distinguishes labels, operators and operands.

SIRIUS-MP is at the "assembly language" end of the trade - its syntax is kept simple, so that a minimal compiler (or hand compilation) will be used to translate it to machine codes, and the semantic interpretations are largely look-ups based on the specific content of the statements coded in a program, with very little variation on certain basic forms for operands and operators.

SPECIFICATION OF SIRIUS-MP:

The specification of a language can be a very formal and very dry process. A language specification is ultimately required in order to clearly convey the meaning of statements coded in the language, the legal variations on such statements, etc. etc. A certain level of consistency in specification is required, for instance, if I want to write a compiler for a given language. At the present time, however, my reasons for formulating SIRIUS are much less demanding than the formal specification of a language: I am interested in creating a method of describing programs which will be heavily commented and used principally for publication in ECS (and possibly other publications.) Thus the specification is left in a fairly "soft" form for the time being within a general framework described in this issue. The time for a formal specification will be the day I sit down and write an appropriate compiler - or a reader decides to do so through impatience and the desire to write one for publication (with the usual royalty of course.)

In lieu of a really formal specification of the SIRIUS-MP language, the next few pages contain an informal description of several notational devices employed in the examples of SIRIUS-MP programs in this issue, and comments on why the forms are used. The areas covered are: STATEMENTS, ADDRESSING & REFERENCE, DATA REPRESENTATIONS, and OPERATIONS. Omitted in the present discussion are several languages forms to be described at a later time, including certain "structured programming" concepts and details of argument/parameter linkage conventions for subroutine calls in SIRIUS-MP.

STATEMENTS :

The basic notational unit of a program which is written in SIRIUS-MP is the "statement." The statement concept embraces the others mentioned on page 5, as can be illustrated by the following prototype format:

```
LABEL:
      TARGET    OP    SOURCE    * COMMENTS    ;
```

As in most decent assemblers, the intent is to make the statement "free form" and thus requiring no fixed column or line boundaries. Hence the following devices are used as a part of the syntax:

The end of a statement is indicated by a ";" (semicolon) as in a host of PL/1-like languages.*

A label, if present, is distinguished from the first (TARGET) operand or the operation mnemonic (OP) by a ":" (colon). With this choice of trailer, labels must not duplicate any operation codes (OP) which can have similar endings.

An asterisk (*) is shown as a separator between the main part of the statement and the comments field at the right.

For examples of the use of this format, see the several program listings included with this issue below. The fields in this prototype statement are as follows:

LABEL - this field (and its ":" separator) is optional and is used to define a symbolic program label. A label is ultimately required to define all symbols used in a program with the exception of certain implicitly defined symbols such as CPU registers and flags.

TARGET - this field (optional) specifies a symbolic reference or absolute address for the memory location(s) or I/O devices which will receive data as a result of an operation. Certain operations will not require a target field for proper notation.

OP - this field is required in order to specify an "operation" to be performed at some time. Certain operations will correspond to executable code in the translation. Others will be used to reserve storage and indicate aspects of the program generation process.

SOURCE - this field is required to specify a minimum of one operand for each operation. Its format will vary depending upon the type of operation intended - variations will include various forms of symbolic reference as well as compound forms used to control functions such as "FOR" loop constructs or "IF" statements.

COMMENTS - here the field intent and use is fairly obvious - to explain what is going on it is useful to make notations.

---

* Note: The alternate form of statement boundary indication to the ";" is to start a new statement on a new line. The examples in this issue all omit the ";" specified above - a detail to be corrected in future issues.

ADDRESSING AND REFERENCE:

For those individuals who have experience with high level languages (eg: FORTRAN, COBOL, PL/1, ALGOL, BASIC etc.) the common experience is to blithly go ahead and program an application with the various "variables" declared within a program by implicit or explicit means. This approach is appropriate for a high order language in most instances because the problem of addressing and referencing data in the computer has been solved in a fairly general and quite reliable manner by the compiler writers. When the time comes to drop down one level of abstraction to the assembly level, the problem of addressing has to be again considered in a more explicit manner since many more details of machine architecture are inherent in such programming. In deciding what forms of addressing and data reference to include in SIRIUS-MP, the low level approach is augmented by several methods of more abstract reference. The following are some key referencing concepts:

ABSOLUTE ADDRESS: The concept here is of a fixed location in the memory address space of the computer or a given I/O instruction channel designation. In a system built around a Motorola 6800 for example, most I/O operations will be carried out with reference to absolute addresses for the I/O interface memory locations - at least in simple programs this will be the case. In the INTEL or National IMP-16 architectures explicit choices of I/O channel require designation of numbers, often in an absolute form.

EXAMPLE: The Octal expression 020023 could represent an absolute address.

SYMBOLIC ADDRESS: The concept here is to reference the name of a data item in an instruction rather than its actual address. In principal all such names map into a fixed and unique address at execution, either through the operation of a compiler's address resolution or through a run time lookup mechanism such as the SYM routine used in the previously published ECS 8008 software. In SIRIUS notation, a symbol is defined by its appearance as a LABEL of a statement, or its existence as a predefined entity such as a register designation .

EXAMPLE: Given label ANYSYM, a reference in some other (eg: assignment) statement might be:

ANYSYM =: 0 (as the TARGET operand.)

INDEXED SYMBOLIC ADDRESS: The concept here is to reference the starting location of a block of memory by the first symbol involved, and to indicate an offset (from zero up) in bytes by a second symbol or literal in parentheses following the first. Thus:

ANYSYM(OFFSET) is a reference to the location ANYSYM plus the current value of OFFSET when the statement is executed.

or

ANYSYM(23) is a reference to address ANYSYM plus 23.

An alternate form of expression for this would be to show an addition (+) operator rather than use a FORTRAN or PL/1- like subscript reference with parenthesis.

SPECIAL SYMBOLIC ADDRESSES: Here the concept is the notation of certain symbols with a fixed meaning, which in an assembler would effectively become "reserved" symbols not subject to redefinition. The forms used in the listings in SIRIUS in this issue are the following :

W(ANYSYM) means "the whereabouts of ANYSYM" and is the notation used to indicate a reference to the absolute address of the symbol.

M(ANYSYM) means "memory reference to the location found in the value of ANYSYM." This is the basic "pointer" form used, and will assume that the value in ANYSYM is a full address (eg: 16 bits for most machines.)

T(ANYJMP) means "the address portion of a jump instruction at ANYJMP". This notation was introduced to allow the equivalent of a FORTRAN assigned GO TO to be used by altering a jump instruction.

A, B, C, D, E, H, L are symbols used freely to represent registers on the Intel 8008 and 8080 type of machine architectures. In translating this reference to a Motorola 6800 or National IMP-16, or other computer architecture, an appropriate software equivalent would be used if registers are not available.

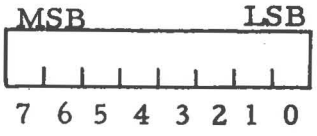
L(ADDRESS), H(ADDRESS) are used to reference the Low and High order portions of a full address (eg: 14 or 16 bits) on typical microcomputers when it is desire to examine only one byte. This is especially useful as a notation for the Intel architectures, but the same functional meaning goes on other machines.

The various forms of addressing and reference described can be used to specify the "operands" - SOURCE and TARGET - of a statement. The concept of a "SYMBOL" is the generalized idea of one of these forms of reference (excluding absolute references.) A "symbol table" for a program is a list of such symbols, usually including some additional information about the item. In a future article on the hand generation of code this concept will be explored in more detail.

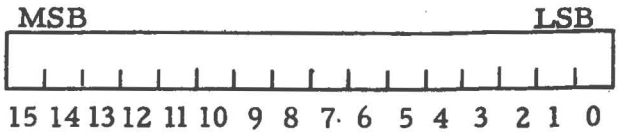
DATA REPRESENTATIONS:

A "data representation" is a method of conceptually treating a group of data bits in the storage of a machine, and is usually fairly dependent upon hardware features of a given machine. The basic data representation of all the extant 8-bit microcomputers is the 8-bit binary integer (two's complement is the rule.) This is augmented in certain machines such as the 8080 and the 6800 by a limited set of 16-bit operations implemented to handle address calculations. For the 16-bit microcomputers and minicomputers, the word length as a rule sets the basic representation as a 16-bit integer, although smaller 8 bit quanta can usually be employed. This immediately suggests that the basic assumption to be built into SIRIUS-MP is that data ought to be operated upon in 8 and 16 bit

quanta. This will prove a useful decision for most processors likely to be in common use by readers of this publication (if there is enough interest, I'll make some comments at a future time on adaptation to 12-bit machines such as the DEC PDP-8 and its imitators.) The two representations are thus (pictorially) ...



8-bit integer



16-bit integer

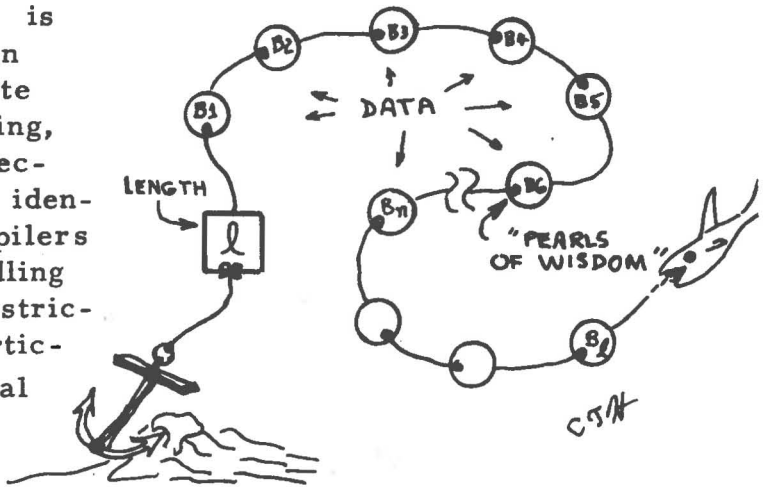
The fact that there are two possible ways to reference integers built into the hardware operations of the typical 8 and 16 bit microcomputer formats, (8008 excluded) leads to a desire to specify a notation for the length of data involved. I could choose among two basic alternatives in this area:

a. Specify data type in some form of declaratory way. This would be analogous to an XPL statement such as "DECLARE X FIXED;" or a FORTRAN statement such as "INTEGER X".

b. Specify data type(length) as a part of the choice of operands used. Here the information on length of operations is specified when the data is used - thus the program has a bit of extra redundancy in its notation (the extra characters needed to specify this type information) but the operations performed are much more visible at the local level.

The choice I made was for the second alternative, primarily to reduce the need for a symbol table to the barest minimum of information - consistent with the simplifications needed for a compact assembler or hand compilation. A secondary reason is the one stated in "b" - local type indications give a better documented program. In the integer operations used by programs in SIRIUS, a single colon (as in "AND:") is used to indicate where an 8-bit operation is involved, and a double colon (as in "AND::") is used to indicate the 16-bit form of an operation. A final comment on integers: where a signed integer representation is required in two's complement notation, the sign of the number is represented by the most significant bit (bit 7 of length 8 words, bit 15 of length 16 words.) This is the bit tested by the "S" flag on the various microcomputers.

Byte String Data: One additional data type will be required for programming the various microcomputers using SIRIUS-MP. This data type is the generalized concept of a "byte string." The representation is designed for manipulation of blocks of data in memory, in a form consisting of a length byte at the "anchor" (starting address) of the string, followed by from 0 to 255 data bytes at consecutive addresses. This is a format which is identical to that used in many byte oriented compilers (eg: XPL) and is a virtual necessity for handling character texts. Applications will not be restricted to character texts, however, for one particular use could include variable length decimal arithmetic using packed BCD byte strings.



Byte strings are most conveniently handled on computers which have byte addressability of memory locations - eg: the IBM 360/370 series as well as the smaller (8080, 8008, 6800) microcomputers. For 16 bit minicomputers and microcomputers, the concept is still useful, but requires explicit address calculations as a part of unpacking and manipulating two bytes per word. Operations on byte strings will use the notation of a number sign "#" to indicate the variable number of bytes involved.

OPERATIONS:

With the above introduction regarding data representations, it is now possible to consider the basic operations possible. The list here represents those used in the notation of the programs in this issue. In a later issue I'll expand the explanations of some of these operations and corresponding machine code for typical machines. There are also several operations which I have not used in the notation of the current set of programs, but which will be the subject of future notes in this area. The following is a list of the operations used with program notation in this issue, omitting the type indicators :

AND	GOTO	INPUT
Assignment(=)	HALT	IOEXCH
CALL	IF	KEYWAIT
CLEAR	IFNOT	OR
DECR	INCR	OUTPUT

The operations AND, OR, GOTO, HALT, INPUT and OUTPUT all have direct analogs in the CPU operations when 8-bit quantities are used with machines such as the 8008, 8080 or 6800. The examples' 8008 generated code versions illustrate one such representation. Some further notes will help illuminate the code generation process for the other operations.

For all operations which have direct analogs in the machine architecture, the code used for the machine level version must consist primarily of establishing the addressability of operands (source and target) and then execution of the operation. This process is illustrated in the several examples. For 8 bit machines with 16 bit operations, the code generated must be generalized to 16 bits - for the 8008 this is done in the illustrated programs by appropriate subroutines for increment, decrement and comparison, so code generation consists of writing down machine codes for a subroutine call and argument linkage.

Assignment always will map into a sequence of operations needed to move data from the source to the target. The 8008 generated code of these examples is an extension of the previously described symbol table mechanism for address lookup (see February 1975 ECS.) For 16 bit quanta this process can often be done using a CPU register pair for the 8 bit machines, but will invariably require a subroutine when byte strings are involved.

The IF statement form used in the examples is found in both a negative and positive sense. In either case the TARGET (lefthand) operand is the place where execution will go if the condition tests true. Two forms of the condition (SOURCE) operand are used:

a. Flag Reference: Here the intent is to use a mnemonic key word, for example "ZERO" to reference one of the CPU flags of a typical micro after an instruction which might alter such flags.

b. Tests: Here the intent is to specify two operands symbolically which are to be compared. I have grouped such references in parenthesis to simplify mechanical interpretation by a compiler, and have used the assignment symbol "=" with its length code with the usual duplicity to indicate the comparison test operation.

A disclaimer is appropriate at this point - I am not satisfied with the IF condition test format illustrated in these examples of several programs, and will be experimenting with some alternatives.

GENERATION OF CODE:

The semantic intent of the language forms used to represent the several programs in this issue can be deduced from the comments in the listings and the general descriptive information in the previous pages. One remaining problem is the generation of code. For the time being, I am limiting information on this (very large) subject to the examples illustrated below for an 8008 case and the notes accompanying the examples. I think there is sufficient information content to facilitate interpretation and generation of corresponding machine code for processors such as the 8080 (very close) or the 6800.

BOOTER: AN "EMERGENCY" BOOTSTRAP LOADER

The first example of a SIRIUS-MP program is a short and self-contained program called "BOOTER." All programs ultimately solve problems. This particular program solved a problem which I had one weekend, and served as an "acid test" of the utility of the ECS-8 tape interface. As soon as I had the interface software up and running (the dump portion presented in March ECS's pages) I began dumping the entire CPU software load to cassettes at regular intervals as a "failsafe" against Boston Edison's next power failure. The planning for that contingency - which by the way did happen in an ice storm in January to my consternation - paid off in a different way: I made the foolish mistake of turning off the power via a switch on my bench, now taped over solidly. Since I was working on SIRIUS-MP as a program writing tool, I took the opportunity to test out the expression it provides by writing the BOOTER source program appearing at the top of the next page. I won't claim perfection, however the original form of the program was essentially the same as the listing illustrated.

Loading is accomplished as follows: in the tape format described in the last issue, the first legitimate data is the length code (two bytes which I knew had "007" and "377" values for my tapes.) Since none of the tape spacing and preparation routines of the IMP program would be available in the blank computer memory being bootstrapped, the only way to synchronize tape data with the program was to listen continuously for the "007" character (state 1, LOOKFIRST tests for "007"), then check for a succeeding "377" byte (state 2, WELLMAYBE tests for "377"), then commence loading bytes starting at

The BOOTER program, listed in SIRIUS-MP...

```
    BOOTER:
1       B           =:          1           * INITIAL STATE IS 1
2       X           =::         2000        * (INTELESE 004/000) START ADDR
3       36          OUTPUT      377         * TURN ON A DISPLAY
4                   CLEAR       A           *
5       A           IOEXCH      4           * RESET THE IO UNIT
    BLOOP:
6       A           =:          27          * "0001 01 1 1" UNIT CONTROL
7       A           IOEXCH      4           * CHECK STATUS OF TAPE
8       A           AND:        140         * MASK OFF RDY & RDA BITS
9       BLOOP       IFNOT       (A=:140)    * LOOP BACK UNTIL READY
    GETCHAR:
10      M(X)        INPUT       2           * READ THE DATA (NO EXCHANGE)
11                  DECR:       B           *
12      LOOKFIRST   IF          ZERO        * HAVE STATE 1 DETECTED
13                  DECR:       B           *
14      WELLMAYBE   IF          ZERO        * HAVE STATE 2 DETECTED
15                  DECR:       B           *
16      FORSURE     IF          ZERO        * HAVE STATE 3 DETECTED
17                  HALT                    * (OOPS! SHOULDN'T GET HERE)
    FORSURE:
18      36          OUTPUT      M(X)        * WRITE TO DISPLAY
19      37          OUTPUT      L(X)        * LOW ORDER ADDR TO DISPLAY
20                  INCR::      X           * POINT TO NEXT BYTE IN MEMORY
21      B           =:          3           * RESET STATE 3 INDICATION
22                  GOTO        BLOOP       * BACK FOR MORE INDEFINITELY
    LOOKFIRST:
23      B           =:          1           * DEFAULT STATE 1 CONTINUE
24      BLOOP       IFNOT       (A=:007)    * LOOK FOR OCTAL "007")
25      B           =:          2           * IF FOUND, STATE SET TO 2
26                  GOTO        BLOOP       * AND GO BACK TO FIND "377")
    WELLMAYBE:
27      B           =:          1           * DEFAULT BACK TO STATE 1
28      BLOOP       IFNOT       (A=:377)    * LOOK FOR OCTAL "377"
29      B           =:          3           * MAIN LOAD LOOP IF FOUND NOW
30                  GOTO        BLOOP       *
```

Variables

A : CPU register for I/O
B : CPU register or mem.
X : Address pointer (CPU)
ZERO : CPU flag for zero result

Notations

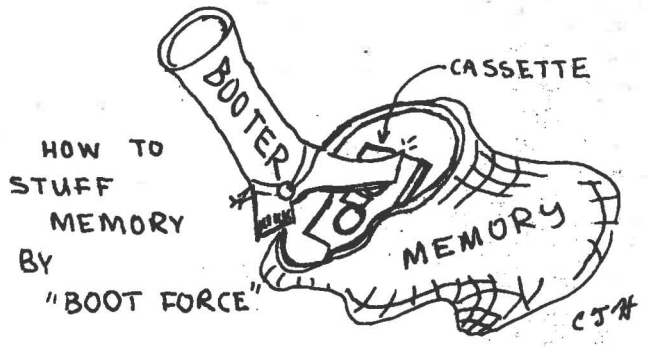
M(X) : memory at location in pointer variable X.

L(X) : low order 8 bytes of X

And the equivalent 8008 version of this algorithm....

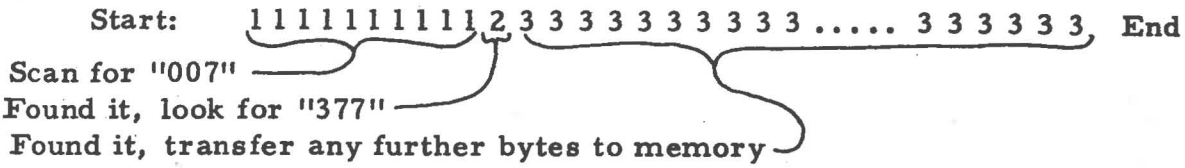
Label	8008 Code Bytes		SIRIUS-MP Statement
BOOTER:	00 \110 = 016	LBI	s 1.
	00 \111 = 001	1	
	00 \112 = 056	LHI	s 2.
	00 \113 = 004	h(LOAD POINT)	
	00 \114 = 066	LLI	
	00 \115 = 000	l(LOAD POINT)	
	00 \116 = 006	LAI	s 3.
	00 \117 = 377	377	
	00 \120 = 175	OUT36	
	00 \121 = 250	XRA	s 4.
	00 \122 = 111	IN4	s 5.
BLOOP:	00 \123 = 006	LAI	s 6.
	00 \124 = 027	"0001 01 11"	
	00 \125 = 111	IN4	s 7.
	00 \126 = 044	NDI	s 8.
	00 \127 = 140	"01 100 000"	
	00 \130 = 074	CPI	s 9.
	00 \131 = 140	"01 100 000"	
	00 \132 = 110	JFZ BLOOP	
	00 \133 = 123	L	
	00 \134 = 000	H	
	00 \135 = 113	IN5 (Read Tape)	s 10.
	00 \136 = 370	LMA	
	00 \137 = 011	DCB	s 11.
	00 \140 = 150	JTZ LOOKFIRST	s 12.
	00 \141 = 166	L	
	00 \142 = 000	H	
	00 \143 = 011	DCB	s 13.
	00 \144 = 150	JTZ WELLMAYBE	s 14.
	00 \145 = 202	L	
	00 \146 = 000	H	
	00 \147 = 011	DCB	s 15.
	00 \150 = 150	JTZ FORSURE	s 16.
	00 \151 = 154	L	
	00 \152 = 000	H	
	00 \153 = 377	HALT	s 17.
FORSURE:	00 \154 = 307	LAM	s 18.
	00 \155 = 175	OUT36	
	00 \156 = 306	LAL	s 19.
	00 \157 = 177	OUT37	
	00 \160 = 055	NEXTA	s 20.
	00 \161 = 016	LBI	s 21.
	00 \162 = 003	3	
	00 \163 = 104	JMP BLOOP	s 22.
	00 \164 = 123	L	
	00 \165 = 000	H	

Label	8008 Code Bytes		SIRIUS-MP Statement
LOOKFIRST:			
	00 \166 = 016	LBI	s 23.
	00 \167 = 001	1	
	00 \170 = 074	CPI	s 24.
	00 \171 = 007	7	
	00 \172 = 110	JFZ BLOOP	
	00 \173 = 123	L	
	00 \174 = 000	H	
	00 \175 = 016	LBI	s 25.
	00 \176 = 002	2	
	00 \177 = 104	JMP BLOOP	s 26.
	00 \200 = 123	L	
	00 \201 = 000	H	
WELLMAYBE:			
	00 \202 = 016	LBI	s 27.
	00 \203 = 001	1	
	00 \204 = 074	CPI	s 28.
	00 \205 = 377	377	
	00 \206 = 110	JFZ BLOOP	
	00 \207 = 123	L	
	00 \210 = 000	H	
	00 \211 = 016	LBI	s 29.
	00 \212 = 003	3	
	00 \213 = 104	JMP	s 30.
	00 \214 = 123	L	
	00 \215 = 000	H	



the known load point (location 2000_8 = intelese 004/000) as initialized at the beginning of the program.

The program is a "state driven" algorithm which has 3 states of execution set by the content of the variable "B" (which maps into a register in the generated code for a microcomputer such as the 8008 code illustrated.) The sequence of states during execution of the main loop "BLOOP" during normal execution is as follows:



The program is set up so that if a false synchronization pattern is detected ("007" followed by any byte other than "377") the "WELLMAYBE" branch of the loop concludes "maybe not" and goes back to scanning the input. The reason for scanning in this manner is to enable the program to be started via an interrupt, after which you can turn on the manual controls of the tape drive confident that the invalid data produced by the MODEM/UART combination during the leader and start up periods will not be falsely interpreted as good data - the specific 16-bit pattern of two bytes involved is not likely to occur due to random noise.

The 8008 code corresponding to the BOOTER program's SIRIUS-MP notation is shown at the bottom of page 12 with symbolic notations of labels, mnemonic op codes and reference numbers to the SIRIUS-MP statements in the listing at the top of the page. The specific hardware assumptions used for this code are documented in previous ECS issues and are not repeated in detail here. For this simple program, the "X" data quantity (a memory pointer) is translated as the content of the H and L register pointer of the 8008. One of the restart routines defined in January ECS is utilized by the generated code - "NEXTA" calculates the next address in H and L. On an 8080 this could be performed without a subroutine using the INX instruction with H and L selected. On a 6800 the corresponding function would be performed using its INX instruction, with the variable X assumed to signify the index register "X".

BOOTER uses output instructions directed at a binary display to illustrate the progress of the program. At initialization, the display left half (OUT36) is loaded with 8 "on" bits. (SIRIUS statement 3). Then, following the synchronization detection, the data transfer branch FORSURE displays the current byte at left (OUT36, statement 18) and the current low order address at right (OUT37 generated by statement 19).

The small loop from statements 6 to 9 is used to cause the program to wait until the flags of the UAR/T subsystem (see article ECS-6 and January 1975 ECS) indicate that a character has been received. The tape unit control code "027_8" defined at statement 6 is used to signify the data rate ("0001" for 1210 baud), channel ("01") and selection for input (the last two bits.)

If you use BOOTER to load IMP from one of the cassettes supplied by M. P. Publishing Co. ($7.50 each post paid) you will have to additionally load by hand the content of the other restart instructions routines before changing the interrupt branch to point to the IMP entry point at location 013/000 (Intelese.)

IMP EXTENSIONS FOR TAPE INTERFACE CONTROL (Continued...)

In the March issue of ECS, I started a presentation of extensions to the Interactive Manipulator Program for tape block write, compare and read operations. This article contains the remainder of the listings. With the exception of the three routines on this page, the additional 8008 code is given in its SIRIUS-MP form and in absolute octal with mnemonics decoded.

One aspect of the SIRIUS-MP language which I have not dealt with explicitly in this issue's discussion is that of argument/parameter linkage for subroutine calls. Because a machine-dependent argument/parameter linkage is used for the 8008 versions of the three routines on this page, I present them here in the same commented listing form used for previous issues of ECS. The routines are utility functions for the two-byte increment/decrement functions and comparison. The parameter linkages to these routines are formed by passing symbols (see Feb. '75 ECS) in registers for lookup.

D2B is the two byte decrement operation, which is entered with the symbol of the operand contained in the 8008's A-register. The operand is decremented by subtraction due to the properties of a zero underflow (the Zero flag detects this state one number too early at 0, not -1.) On return, the carry flag indicates a 16-bit underflow if any

I2B is the corresponding two byte increment operation, which is also entered with the symbol of the operand in the 8008's A register. The 8008's increment instructions are used, since the zero state is a reliable overflow indicator. On return, the zero flag indicates a 16-bit overflow if any.

C2B is a two byte comparison operation, with a more complicated linkage. The two operands are passed as symbols in the B and C registers. The result is passed back as the content of the "E" register : 1 if not equal, 2 if equal. This can be tested by a decrement instruction followed by a jump on zero .

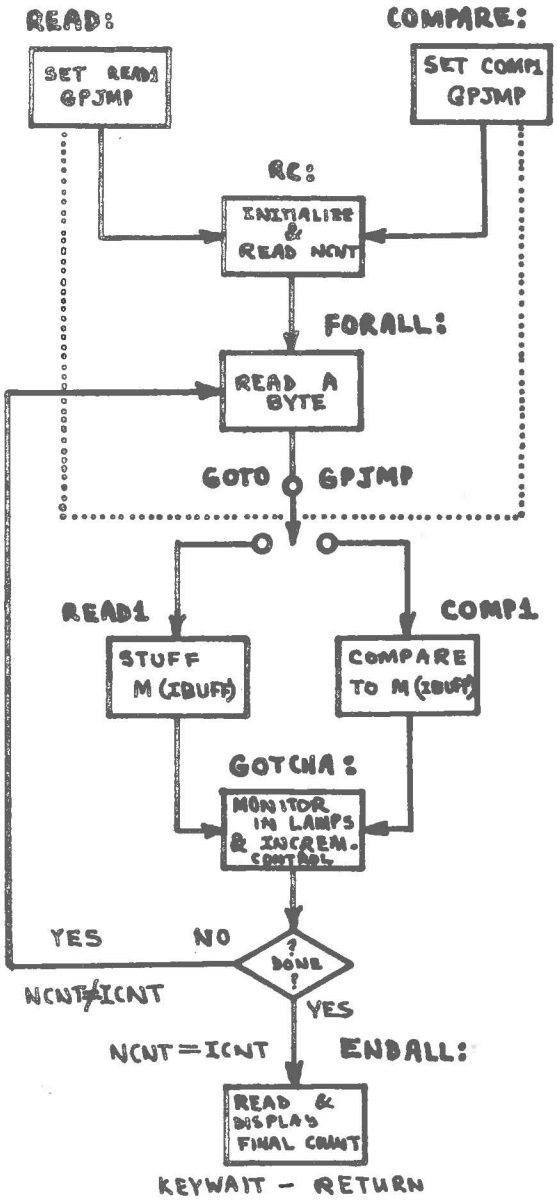
```
D2B:
        012\132 = 075 SYM       Go pick up argument address
        012\133 = 060 INL       Point ahead (assume not at page bound)
        012\134 = 307 LAM       Fetch the low order byte.
        012\135 = 024 SUI       Subtract 1 - decrement will not do!
        012\136 = 001 1
        012\137 = 370 LMA       Save result
        012\140 = 003 RFC       Return on no borrow condition.
        012\141 = 061 DCL       Point to high order byte
        012\142 = 307 LAM       Fetch it
        012\143 = 024 SUI       Also decrement with subtract
        012\144 = 001 1           so that borrow (C) may be set...
        012\145 = 370 LMA       Save result
        012\146 = 007 RET       With carry indicating net underflow.

I2B:    Routine to increment two bytes - enter with symbol parameter in A
        011\313 = 075 SYM       Look up the parameter address
        011\314 = 060 INL       Point to,
        011\315 = 317 LBM         load from memory,
        011\316 = 010 INB           increment and
        011\317 = 371 LMB             save the low order byte.
        011\320 = 013 RFZ       Return direct if no overflow
        011\321 = 061 DCL       Point to,
        011\322 = 317 LBM         load from memory,
        011\323 = 010 INB           increment,
        011\324 = 371 LMB             and save the high order byte.
        011\325 = 007 RET       Then return always.

C2B:    Routine to compare bytes - in two's. Enter with symbol parameters in
        010\234 = 046 LEI                       registers B and C.
        010\235 = 001 1         Return default 1 (not equal.)
        010\236 = 301 LAB
        010\237 = 075 SYM       Fetch first parameter address
        010\240 = 337 LDM         and fetch the parameter.
        010\241 = 302 LAC
        010\242 = 075 SYM       Fetch second parameter address
        010\243 = 303 LAD         and compare against
        010\244 = 277 CPM           first parameter value...
        010\245 = 013 RFZ       Return (E=1) if unequal.
        010\246 = 055 NEXTA     Point to next address of second parm.
        010\247 = 337 LDM       Fetch second parm second byte
        010\250 = 301 LAB
        010\251 = 075 SYM       Point to first  parm again
        010\252 = 055 NEXTA       look NEXTA him too!!!
        010\253 = 303 LAD
        010\254 = 277 CPM       Compare first parm, second byte
        010\255 = 013 RFZ       And again return (E=1) if unequal.
        010\256 = 046 LEI       Otherwise both bytes of both
        010\257 = 002 2           two sets are equal and can
        010\260 = 007 RET         return with equality result.
```

The notational power of a more abstract method of programming is illustrated by comparing the expression of the new IMP extension segments on page 16 with the corresponding "generated code" for the 8008 printed later. The routines listed in SIRIUS-MP form for the tape extension begin with the main portion of the program...

READ/COMPARE main routine is at the left hand side of page 16 held sideways. This 33-statement SIRIUS-MP program is invoked when the IMP command decoder detects a "shift R" for read or "shift C" for compare. The difference in the two routines is determined by the entry point - line 1 for READ, line 28 for COMPARE. The logic at the entry points sets up a jump address in the "GPJMP" indirect branch location (this overwrites the previous use of GPJMP to get to READ or COMPARE from IMP.) This switch (the choice of branch paths) is required so that the same general control flow can be use for both the READ and COMPARE operations - the difference being in what is done with the information read from tape. The switch point in the flow occurs at statement 14, and can be illustrated in flow chart terms by the diagram at the right.



The common portion of the program provides the overall structure of a read operation: initialize the UAR/T, read a dummy character at the first RDA time, read the two length code bytes written by the OUTCNT routine (see below) when the tape is prepared, then enter a loop which continues until the data count is exhausted.

When the READ1 branch of the flow is taken during a read operation, the current memory location pointed to by IBUFF receives the input character found in a variable called "B" (a CPU register for the 8008 version of the program.)

When the COMP1 branch of the flow is taken during a compare operation, the current byte pointed to by IBUFF is compared to the input byte in the variable "B" - and an error count is incremented in the variable "BADDATA" (16 bits worth) to keep a tally of the badnesses.

The data count is kept in the variable "ICNT" which starts out at -1 and is counted up until it equals the block count stored in "NCNT" after it is read from the tape. The test for end of transfer is found at statement 20, a SIRIUS "IFNOT" operation.

IMP program tape extensions expressed in a SIRIUS fashion...

```
    READ:
1       T(GPJMP)    =::     W(READ1)    * SET READ JUMP SWITCH
    RC:
2       TAPECTRL    OR:     "0000 00 1 1"   * FORCE INPUT SELECT
3               CLEAR   A
4       A           IOEXCH  4           * RESET THE IO UNIT
    INITIALIZE:
5       4           OUTPUT  TAPECTRL    * SET SELECTED CONTROL STUFF
6       IBUFF       =::     MEMADDR     * START INPUT AT MEMADDR
7       ICNT        =::     -1          * INITIAL COUNT TO MATCH OUTPUT
    DUMMYIN:
8               CALL    INPUT2      * GO FETCH BYTE (WAIT LOOP)
    HIGHLNGTH:
9               CALL    INPUT2      * GET HIGH ORDER LENGTH
10      NCNT        =:      B           * SAVE B INPUT IN NCNT H.O.
    LOWLNGTH:
11              CALL    INPUT2      * GET LOW ORDER LENGTH
12      NCNT(1)     =:      B           * STORE AT NCNT+1
    FORALL:
13              CALL    INPUT2      * NORMAL DATA BYTE FETCH
14              GOTO    GPJMP       * SELECT COMPARE OR READ VIA
                                    *    VARIABLE JUMP TARGET
    READ1:
15      M(IBUFF)    =:      B           * IF READ THEN STORE IT
    GOTCHA:
16      37          OUTPUT  B           * DISPLAY INPUT DATA
17      36          OUTPUT  0           * CLEAR OTHER DISPLAY TO ZERO
18              INCR::  IBUFF       * POINT TO NEXT INPUT ADDRESS
19              INCR::  ICNT        * INCREMENT WORKING COUNT
20      FORALL      IFNOT   (ICNT=::NCNT)   * TEST END OF BLOCK
    ENDALL:
21              CALL    INPUT2      * READ FINAL LENGTH BYTE
22      36          OUTPUT  B           *    AND DISPLAY
23              CALL    INPUT2      * READ SECOND FINAL LENGTH BYTE
24      37          OUTPUT  B           *    AND DISPLAY IT TOO
25      TAPECTRL    AND:    "1111 11 0 0"   * TURN OFF INPUT SELECT
26      4           OUTPUT  TAPECTRL    * TURN OFF THE DRIVE... PATCH
                                    *    IN A 2 SECOND WAIT HERE
                                    *    IF NEEDED - SEE TEXT...
27              KEYWAIT             * SLEEP PERCHANCE TO DREAM
    COMPARE:
28      T(GPJMP)    =::     W(COMP1)    * SET COMPARE JUMP SWITCH
29      BADDATA     =::     0           * ZERO OUT BAD DATA...COUNT
30              GOTO    RC          * ENTER NORMAL FLOW
    COMP1:
31      GOTCHA      IF      (M(IBUFF)=:B)   * TEST TAPE AGAINST MEMORY
32              INCR::  BADDATA     * MISSED SOME BITS!!!
33              GOTO    GOTCHA      * BACK FOR MORE...
```

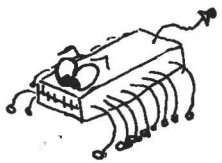
```
    INPUT2:
1       A           =:      TAPECTRL    * FETCH IO CONTROL WORD
2       A           IOEXCH  4           * EXCHANGE FOR STATUS
3       B           =:      A           * SAVE STATUS IN B
4       A           AND:    "01 100 000"    *  MASK DESIRED BITS
5       INPUT2      IFNOT   (A=:"01 100 000")   *  WAIT TILL READY
6       A           =:      B           * RESTORE STATUS FROM B
7       A           AND:    "00 000 111"    * MASK ERROR BITS
8       INPUTIT     IF      (A=:"00 000 111")   *  INVERTED NO ERRORS
9               INCR::  BADFORM     * INCREMENT DATA FORMAT ERRORS
    INPUTIT:
10      A           INPUT   5           * READ THE LATESTCHARACTER
11      B           =:      A           * PASS BACK VIA B REGISTER
12              RETURN              * BACK TO CALLER


    NEWOUTCNT:
1       B           =:      15₁₀        * MAKE IT 1.5 SEC DELAY
2               CALL    WAITCS      *     VIA CENTISECOND DELAYER
3       A           =:      COUNT       * SEND OUT THE FIRST
4       B           =:      A           *     COUNT BYTE
5       5           OUTPUT  A           *       AND SAVE IN B
6               CALL    WAITOUT     * WAIT UNTIL NOT BUSY
7       A           =:      COUNT(1)    * GET SECOND BYTE AT COUNT+1
8       C           =:      A           * SAVE IT IN C
9       5           OUTPUT  A           * AND OUTPUT TO TAPE
10              CALL    WAITOUT     * WAIT UNTIL NOT BUSY
11              RETURN              * THEN BACK


    ONOFF:
1       A           =:      TAPECTRL    * FETCH OLD TAPE CONTROL
2       A           AND:    "00 000 010"    * CHECK OLD STATE OF SELECT
3       TON         IF      ZERO        * CHANGE TO ON IF OFF
    TOFF:
4       B           =:      2           * CHANGE TO OFF IF ON
5               GOTO    EITHER      * THEN DO THE CHANGE
    TON:
6       B           =:      0           * CHANGE TO ON IF OFF
    EITHER:
7       A           =:      TAPECTRL    * FETCH OLD CONTROL AGAIN
8       A           AND:    374         * MASK AND SAVE HIGH ORDER 6 BITS
9       A           OR:     B           * COMBINE WITH NEW CONTROL
10      TAPECTRL    =:      A           * SAVE NEW CONTROL
11      4           OUTPUT  A           * TURN TAPE MOTOR OFF OR ON
12              KEYWAIT             * BACK TO SLEEP YOU IMP!!!!
```

Note: Reference numbers to SIRIUS statements are provided at the local level for each block of functional code illustrated here. They correlate to the 8008 examples of executable machine codes, within each block.

Notations: T(GPJMP) : address part of jump
W(READ1) : mem. address of READ1
NAME(n) : n^{th} byte of NAME



8008 Generated Code for READ/COMPARE routines (p. 16, left)

Label	8008 Code Bytes		SIRIUS-MP Statement #
READ:			
	004\000 = 006	LAI	s 1.
	004\001 = 010	s(GPJMPAL)	
	004\002 = 075	SYM	
	004\003 = 076	LMI	
	004\004 = 107	L(READ1)	
	004\005 = 060	INL	
	004\006 = 076	LMI	
	004\007 = 004	H(READ1)	
RC:			
	004\010 = 006	LAI	s 2.
	004\011 = 014	s(TAPECTRL)	
	004\012 = 075	SYM	
	004\013 = 307	LAM	
	004\014 = 064	ORI	
	004\015 = 003	"00000011"	
	004\016 = 370	LMA	
	004\017 = 250	XRA	s 3.
	004\020 = 111	IN4	s 4.
INITIALIZE:			
	004\021 = 006	LAI	s 5.
	004\022 = 014	s(TAPECTRL)	
	004\023 = 075	SYM	
	004\024 = 307	LAM	
	004\025 = 111	IN4	
	004\026 = 006	LAI	s 6.
	004\027 = 006	s(MEMADDR)	
	004\030 = 075	SYM	
	004\031 = 317	LBM	
	004\032 = 060	INL	
	004\033 = 327	LCM	
	004\034 = 006	LAI	
	004\035 = 020	s(IBUFF)	
	004\036 = 075	SYM	
	004\037 = 371	LMB	
	004\040 = 060	INL	
	004\041 = 372	LMC	
	004\042 = 006	LAI	s 7.
	004\043 = 016	s(ICNT)	
	004\044 = 075	SYM	
	004\045 = 006	LAI	
	004\046 = 377	"11111111"	
	004\047 = 370	LMA	
	004\050 = 060	INL	
	004\051 = 370	LMA	
DUMMYIN:			
	004\052 = 106	CAL INPUT2	s 8.
	004\053 = 061	L	
	004\054 = 012	H	
HIGHLNGTH:			
	004\055 = 106	CAL INPUT2	s 9.
	004\056 = 061	L	
	004\057 = 012	H	
	004\060 = 006	LAI	s 10.
	004\061 = 022	s(NCNT)	
	004\062 = 075	SYM	
	004\063 = 371	LMB	
LOWLNGTH:			
	004\064 = 106	CAL INPUT2	s 11.
	004\065 = 061	L	
	004\066 = 012	H	
	004\067 = 006	LAI	s 12.
	004\070 = 022	s(NCNT)	
	004\071 = 075	SYM	
	004\072 = 055	NEXTA	
	004\073 = 371	LMB	
FORALL:			
	004\074 = 106	CALL INPUT2	s 13.
	004\075 = 061	L	
	004\076 = 012	H	
	004\077 = 006	LAI	Globally optimized code moved ahead of the GPJMP
	004\100 = 020	s(IBUFF)	
	004\101 = 106	CALL MEMSYM	
	004\102 = 002	L	
	004\103 = 012	H	
	004\104 = 104	JMP GPJMP	s 14.
	004\105 = 015	L	
	004\106 = 000	H	

Label	8008 Code Bytes		SIRIUS-MP Statement #
READ1:			
	004\107 = 371	LMB	s 15.
GOTCHA:			
	004\110 = 301	LAB	s 16.
	004\111 = 177	OUT37	
	004\112 = 250	XRA	s 17.
	004\113 = 175	OUT36	
	004\114 = 006	LAI	s 18.
	004\115 = 020	s(IBUFF)	
	004\116 = 106	CAL I2B	
	004\117 = 313	L	
	004\120 = 011	H	
	004\121 = 006	LAI	s 19.
	004\122 = 016	s(ICNT)	
	004\123 = 106	CAL I2B	
	004\124 = 313	L	
	004\125 = 011	H	
	004\126 = 016	LBI	s 20.
	004\127 = 016	s(ICNT)	
	004\130 = 026	LCI	
	004\131 = 022	s(NCNT)	
	004\132 = 106	CAL C2B	
	004\133 = 234	L	
	004\134 = 010	H	
	004\135 = 041	DCE	
	004\136 = 150	JTZ FORALL	
	004\137 = 074	L	
	004\140 = 004	H	
ENDALL:			
	004\141 = 106	CALL INPUT2	s 21.
	004\142 = 061	L	
	004\143 = 012	H	
	004\144 = 301	LAB	s 22.
	004\145 = 175	OUT36	
	004\146 = 106	CALL INPUT2	s 23.
	004\147 = 061	L	
	004\150 = 012	H	
	004\151 = 301	LAB	s 24.
	004\152 = 177	OUT37	
	004\153 = 006	LAI	s 25.
	004\154 = 014	s(TAPECTRL)	
	004\155 = 075	SYM	
	004\156 = 307	LAM	
	004\157 = 044	NDI	
	004\160 = 374	"11 111 100"	
	004\161 = 370	LMA	
	004\162 = 111	IN4	s 26.
	004\163 = 025	KEYWAIT	s 27.
COMPARE:			
	004\164 = 006	LAI	s 28.
	004\165 = 010	s(GPJMPAL)	
	004\166 = 075	SYM	
	004\167 = 076	LMI	
	004\170 = 206	L(COMP1)	
	004\171 = 060	INL	
	004\172 = 076	LMI	
	004\173 = 004	H(COMP1)	
	004\174 = 006	LAI	s 29.
	004\175 = 024	s(BADDATA)	
	004\176 = 075	SYM	
	004\177 = 250	XRA	
	004\200 = 370	LMA	
	004\201 = 060	INL	
	004\202 = 370	LMA	
	004\203 = 104	JMP RC	s 30.
	004\204 = 010	L	
	004\205 = 004	H	
COMP1:			
	004\206 = 301	LAB	s 31.
	004\207 = 277	CPM	
	004\210 = 150	JTZ GOTCHA	
	004\211 = 110	L	
	004\212 = 004	H	
	004\213 = 006	LAI	s 32.
	004\214 = 024	s(BADDATA)	
	004\215 = 106	CAL I2B	
	004\216 = 313	L	
	004\217 = 011	H	
	004\220 = 104	JMP GOTCHA	s 33.
	004\221 = 110	L	
	004\222 = 004	H	

8008 Generated Code for MISCELLANEOUS routines (p 16, right)

```
Label     8008 Code Bytes                    SIRIUS-MP Statement #
INPUT2:
          012\061 = 006 LAI                  s 1.
          012\062 = 014 s(TAPECTRL)
          012\063 = 075 SYM
          012\064 = 307 LAM
          012\065 = 111 IN4                  s 2.
          012\066 = 310 LBA                  s 3.
          012\067 = 044 NDI                  s 4.
          012\070 = 140 "01 100 000"
          012\071 = 074 CPI                  s 5.
          012\072 = 140 "01 100 000"
          012\073 = 110 JTZ INPUT2
          012\074 = 061 L
          012\075 = 012 H
          012\076 = 301 LAB                  s 6.
          012\077 = 044 NDI                  s 7.
          012\100 = 007 "00 000 111"
          012\101 = 074 CPI                  s 8.
          012\102 = 007 "00 000 111"
          012\103 = 150 JTZ INPUTIT
          012\104 = 113 L
          012\105 = 012 H
          012\106 = 006 LAI                  s 9.
          012\107 = 026 s(BADFORM)
          012\110 = 106 CALL I2B
          012\111 = 365 L
          012\112 = 010 H
INPUTIT:
          012\113 = 113 IN5                  s 10.
          012\114 = 310 LBA                  s 11.
          012\115 = 007 RETURN

OUTCOUNT:
          012\200 = 104 JMP NEWOUTCNT   Here is a patch to get to the
          012\201 = 116 L                new version of OUTCOUNT.
          012\202 = 010 H
NEWOUTCNT:
          010\116 = 016 LBI                  s 1.
          010\117 = 017 15(10)
          010\120 = 106 CALL WAITCS          s 2.
          010\121 = 116 L
          010\122 = 012 H
          010\123 = 006 LAI                  s 3.
          010\124 = 022 s(COUNT)
          010\125 = 075 SYM
          010\126 = 307 LAM
          010\127 = 310 LBA                  s 4.
          010\130 = 113 IN5                  s 5.
          010\131 = 106 CAL WAITOUT          s 6.
          010\132 = 147 L
          010\133 = 012 H
          010\134 = 006 LAI                  s 7.
          010\135 = 022 s(COUNT)
          010\136 = 075 SYM                  s
          010\137 = 060 INL
          010\140 = 307 LAM
          010\141 = 320 LCA                  s 8.
          010\142 = 113 IN5                  s 9.
          010\143 = 106 CALL WAITOUT         s 11.
          010\144 = 147 L
          010\145 = 012 H
          010\146 = 007 RETURN               s 12.
```

Patches to Previous Code

```
TAPECMDS:
          012\352 = 317 "O"
          012\353 = 321 L(JONOFF)

          012\272 = 012 "34" is TAPECMDS (new value)
          012\273 = 352
JONOFF:
          012\321 = 104 JMP ONOFF      IMP entry to the
          012\322 = 264 L              ONOFF routine sand-
          012\323 = 011 H              wiched in spare bytes.
READJ:
          013\313 = 104 JMP READ       New IMP READ
          013\314 = 000 L              entry address in
          013\315 = 004 H              this jump.
COMPJ:
          013\316 = 104 JMP COMPARE    New IMP COMPARE
          013\317 = 164 L              routine entry address
          013\320 = 004 H              now in this jump.
```

```
Label     8008 Code Bytes                    SIRIUS - MP Statement #
ONOFF:
          011\264 = 006 LAI                  s 1.
          011\265 = 014 s(TAPECTRL)
          011\266 = 075 SYM
          011\267 = 307 LAM
          011\270 = 044 NDI                  s 2.
          011\271 = 002 "00 000 010"
          011\272 = 150 JTZ  TON             s 3.
          011\273 = 302 L
          011\274 = 011 H
TOFF:
          011\275 = 016 LBI                  s 4.
          011\276 = 000 0
          011\277 = 104 JMP  EITHER          s 5.
          011\300 = 304 L
          011\301 = 011 H
TON:
          011\302 = 016 LBI                  s 6.
          011\303 = 002 2
EITHER:
          011\304 = 307 LAM                  s 7.
          011\305 = 044 NDI                  s 8.
          011\306 = 374 "11 111 100"
          011\307 = 261 ORB "xx xxx xBo"     s 9.
          011\310 = 370 LMA                  s 10.
          011\311 = 111 IN4                  s 11.
          011\312 = 025 KEYWAIT
```

Tape Extension VARIABLES (in order of appearance)

GPJMP, symbol 10

TAPECTRL, symbol 14

A, CPU register

MEMADDR, symbol 06, input to tape transfers.

IBUFF, symbol 020

ICNT, symbol 016

NCNT, symbol 022

B, CPU register

BADDATA, symbol 24

BADFORM, symbol 26

COUNT, symbol 22

ZERO, CPU flag

Note: NCNT, COUNT are equivalent; ICNT and TCOUNT (see March ECS) are equivalent.

The INPUT2 subroutine is at the top right hand side of page 16 held sideways. This 12-statement SIRIUS-MP subprogram is invoked by a subroutine CALL whenever another program wants to "read" a byte from the tape unit according to the content of TAPECTRL. The reading method incorporated in the software of IMP to date is a "polling" technique in which a loop tests status bits of the I/O device (UAR/T "RDA" and a motor turn-on oneshot "ready" signal.) The loop consists of SIRIUS-MP statements 1 to 5 of INPUT2. The routine breaks out of the loop, reads the data and returns with the data byte in the variable "B" (a register in the 8008 generated code). The three UAR/T reception status bits (parity error/framing error/overrun error) are checked and an error count in BADFORM is incremented if no errors are detected.

The OUTCOUNT routine of the March issue of ECS was modified to improve performan in the course of rewriting the comparison software in SIRIUS for this issue. The problem with the original version was the fact that an explicit output wait is required for reliable reading of the data. Thus a patch is placed at location 012/200 to jump to the new version of the program, loaded in some spare memory address space at 010/116. The NEWOUTCNT has two changes: a) I increased the time delay before output to 1.5 seconds (SIRIUS statements 1 and 2); b) I have inserted calls to WAITOUT after each output of a byte (SIRIUS statements 5 and 9 of NEWOUTCNT.)

The ONOFF routine is a new routine added to support a new tape control command, "TO" entered from the keyboard device. The idea here is to have a way to turn on the motor for purposes of listening to data with the ear, for rewinds of long duration, or for recording non-digital comments with the cassette recorder's built-in microphone. The ONOFF routine itself is very simple, comprising a set of 12 SIRIUS statements which map into 23 8008 bytes in the sample generated code. The "TO" function complements the current state of the motor control bit in TAPECTRL and outputs the result to currently selected tape drive via the "IN4" instruction connected to the tape controller.

In setting up to run IMP with the new extensions, the patches to TAPECMDS, JONOFF, and READJ/COMPJ locations of IMP must be made as indicated in the detail listing of page 18. The TAPECMDS table is extended for the new "O" subcommand by starting it one byte earlier; the symbol table symbol "34" for TAPECMDS is adjusted to reflect this addition. The new execution jump JONOFF is added to get the program into the ONOFF routine, and the READJ/COMPJ jumps are changed to reflect altered placement of these routines from the original layout. One other change is required to the symbol table published previously: the address of symbol "20" should be changed to "220" in byte 012/301 of the 8008 code. This symbol has been changed from its original use and now becomes the memory pointer "IBUFF" with two bytes instead of the original 1 byte of reserved space.

COMMENTS ON THE ECS-8 DESIGN:

The output of the TSI (serial data to the computer interface) line is not suitable for an interrupt driven UAR/T software interface without use of some masking logic. The problem is this: the FSK input decode is done by the phase lock loop of the XR-210. When null inputs (eg: tape leader period, or any time without a mark signal) occur, the phase lock loop hunts around for a lock - thus causing the comparator to have its input switch back and forth with the result being a digital noise signal on the TSI line. If the UART is listening, it will decode erroneous characters in this mode. The software of this article ignores the problem by not listening unless good data is coming.

Notes on NAVIGATION IN THE VICINITY OF ∝- AQUILA... #1

This article begins a regular series of information and commentary on the use of the Intel 8080 in an ECS context, with occasional specific reference to packaged systems such as the MITS Altair product. In addition to the MITS product, there is at least one other source of the 8080 chips and boards advertising in the pages of Radio Electronics/Popular Electronics. This first installment concerns some general comments on the 8080 instruction set and specific suggestions concerning 16-bit arithmetic operations (addition/subtraction) in applications other than address calculations.

AQ-1.1: Addressing Modes.

One of the most basic questions to be asked whenever you ponder the use of a new computer instruction architecture is "what are its addressing modes?" The answers all lie in the hardware designer's backyard whenever a specific existing machine such as the 8080 is considered. How do I get at the data in memory when I want to perform some operation in the machine? Are there different ways of reaching the same data item? And so on. The effects of addressing and data reference will color the whole process of generating programs for the architecture of the machine in question. For instance, if the machine is a "stack machine" (not a machine with a stack, but one designed for operations between stack elements) then the addressing can almost exclusively be implied by the way operations are done. On such a machine, the only bits needed for an instruction are the data bits which specify an operation. But in the real world of existing and implemented machines available to the ECS type of application, the coloring of coding is much more conventional - addressing is performed as part of the instruction or as part of an implied setup in a CPU register under program control. In the Intel 8080 (as in the 8008) the design of addressing modes is a fairly arbitrary pot-pouri of methods fraught with special cases not ammenable to concise summary without losing information. In order to write programs these addressing modes must be known and understood so that the best of alternatives (if any) can be evaluated and used in a given programming situation. In the comments below, a few of the conventional addressing concepts in computer designs are isolated and illustrated with regard to the 8080.

AQ-1.2: Immediate Addressing.

Immediate data addressing exists in some form in most contemporary computers, with the usual definition being a constant bit pattern of one word length, following the operation code in a program. The 8080 includes this form of addressing with all the immediate operations which exist on its antecedent the 8008, plus some extensions which make the architecture more useful as a general purpose computing element. The primary extension of immediate addressing is to the inclusion of a long (16-bit) form of the concept in certain limited classes of move (load/store) operations with respect to CPU registers. The 8080 partitions 6 of the 7 CPU registrs into three pairs "index registers" which may be loaded with 16-bit numbers using immediate addressing. The primary intention of such operations is the loading of an address, but programmers can and

do use operations for whatever purpose is required to solve a problem - so whenever one needs a 16-bit "literal" data item this form of double byte immediate operation can be used to load CPU registers.

One particular use of the two-word immediate form in its intended application is the initialization of the stack pointer as a part of setting up execution of a program. In large scale systems the equivalent of a stack pointer (ie: system defined addressing parameters) is usually determined by the "operating system" prior to the call which invokes a user-program. But in your use of a microcomputer of the 8080 (or Motorola 6800) design, with minimal software, you can make no assumptions about the initialization. To be used, the stack must exist in random access read/write memory so that the temporary linkage data associated with the CALL operation and its arguments can be stored. In order for this linkage to occur, the stack pointer (SP) must point to the RAM area. One way to initialize the stack pointer following the start of execution is contained in the following SIRIUS-MP notation and its 8080 translation:

SIRIUS:	8080:
SP =:: location	LXI SP, location

In both instances, the "location" is the 16-bit integer number which is the address of the stack area.

AQ-1.3: Absolute Addressing.

The design of a computer instruction set involves many trade-offs, the evaluation of options with inputs ranging from the preferences of programming individuals to the physical constraints of the LSI chip. In the best of all possible programming worlds, one would like to see a consistent set of addressing modes applicable in principle to any of the basic operations possible. In particular, a more extended use of an absolute (in-instruction stream) form would be desirable than has been implemented with the 8080. There are two basic operations available in the 8080 instruction set which reference memory from within the instruction stream. These are the load (LDA, LHLD) and store (STA, SHLD) operations in 8 and 16 bit variations. For program code which involves fixed data areas at locations allocated by hand or by an assembler/compiler, these operations will be used extensively to prepare data for the execution of actual "work" -since the actual work cannot reference memory directly. The use of load and store for this purpose is highly conventional in many minicomputers, although usually at least one of the algebraic/logic operation operands can be acquired by a direct or indirect memory reference in the instruction stream. (As a point of contrast, the Motorola 6800 microcomputer can perform most of its arithmetic/logical operations with one in-instruction address reference to memory.)

AQ-1.4: Pointer Addressing.

One area where the 8080 has some excellence is in the number of CPU registers it has and the fact that three different pairs can be used as "index registers" for fetching

data to an accumulator (all pairs) or referencing memory operands (H/L only) of the arithmetic operations. It is thus fairly easy to keep pointers around locally in the CPU without the need to transfer them to another location when making a reference based upon the index. The pointers are, however, only good for one operation in general - referencing data in load/store situations, and thus not as useful as they might otherwise have been. The memory reference modes of all the 8-bit arithmetic and logical instructions use one of these pointers, the H/L register pair, to address the one memory operand (the implied second operand is the accumulator register A.) All the procedures and tricks applicable to setting up H/L pointer addresses in the earlier 8008 microcomputer design apply as well to the equivalent H/L forms of the 8080.

One particular programming trick which will prove useful in manipulating blocks of data involves the use of one pointer pair - D/E - to point to one operand block and a second pointer pair - H/L to reference the second block. Suppose the problem is to "AND" all the bytes of one block with the bytes of another and to store the result in the second. The basic set of instructions used to set up the loop would be:

```
LXI D   address 1
LXI H   address 2       set up addresses
```

With this setup, the heart of a loop to transfer the data with an AND condition as required by the problem statement would be:

```
MOV, A, M       Fetch first operand byte
XCHG            Establish second operand address, but
                   save first operand address
ANA  M          AND with second byte
MOV  M, A       Save in second operand byte
INXH            Increment address
XCHG            Move back in exchange
INXH            Increment address
```

This code does not include the instructions needed to establish a loop - to transfer a block with this operation would require a loop count and loop count decrement followed by conditional test for continuation.

This same general scheme of switching the D/E with H/L registers can be used quite widely your program must step simultaneously through two regions of memory. The technique only works with D/E & H/L unless you want to take a calculated risk and exchange with the stack pointer instead of D/E.

AQ-1.5: 16-Bit Operations & 16-Bit Addition/Subtraction.

The 8080 has a specific and limited set of 16-bit operations which can be used to some advantage both for the intended purpose (address calculation and setup) and in more general problems. The 16-bit operations are ...

16-bit Load and Store between register pairs and memory or immediate (Load only) data.
16-bit Addition intended for address calculation.
16-bit Increment/Decrement useful in loop counting & address changing.

For the more general usage of the 16-bit addition operation in programs requiring the extended precision addition / subtraction, the H/L register pair can be treated as if it were a 16-bit accumulator for the purposes of calculation with the actual results being stored ultimately in memory operands. The boxes below illustrate two calculations in 16 bit precision, under the following assumptions:

a. Variable P is a two-byte operand at locations P and P + 1.
b. Variable Q is a two-byte operand at locations Q and Q + 1.
c. The content of A, H and L registers is irrelevant prior to and following the calculation.
d. Absolute addressing will be used with the result stored back in P, as if P were a "software accumulator."

Note the differences in the size of the little routines involved - for the addition case, the setup and execution is fairly compact. For subtraction the need to form the two's complement negative of the Q operand complicates the picture...

The SIRIUS-MP statement: P +:: Q * 16-BIT ADD generates...

```
LHLD    Q     Get first operand bytes to Q
XCHG          Move first op to D/E
LHLD    P     Get second operand (soft. accum.)
DADD          Add Q to P giving P
SHLD    P     Store result back into new P value
```

The SIRIUS-MP statement: P -:: Q * 16-BIT SUBTRACT generates...

```
LDA     Q     Get first byte, negative operand.
CMA           Complement it
MOV     D, A  Move it to D of D/E pair.
LDA     Q+1   Get second byte, negative operand.
CMA           Complement it.
MOV     E, A  Move it to E of D/E pair.
INX     D     Increment complement giving -Q value
LHLD    P     Get software accumulator value
DADD          Value of P - Q now in H/L
SHLD    P     Save back in software accumulator.
```

After either of these operations, the carry flag can be tested to find out if an overflow occurred, thus in principal allowing extended precision of greater precision than 16 bits.

One particular 16-bit operation may prove of use in certain contexts. This is the 16-bit addition of the H/L register pair to itself by means of the DADH instruction. There are two instances where this variation of 16-bit addition stands out for potential utility:

a. Suppose I want to address an extended array of data kept in 2, 4, 8 or 2^n byte quanta. The shift properties of this addition (it multiplies H/L by 2) can be used "n" times to modify an integer array index ala FORTRAN or PL/1 into a useful address offset.

b. This left shift operation can form the basis of an integer multiply operation.

AQ-1.6 A Ceremonial "Nit":

It serves no good end to act the part of a contentious critic, but... at the risk of being in the position of a pot calling the kettle black I do protest MITS' use of the Anquish Languish (technical dialect) in the Altair 8800 manual I examined recently:

> Implement: This verbalized noun is conventionally used in technical contexts such as "to implement a system." (Ie: to create the system.) A computer designer implements an LDA or STA instruction; the programmer codes said implemented instruction (ie: selects it) as part of his own process of implementing a software system. Programmers never use unimplemented instructions as a matter of course. (If you take Webster literally one might come out with the MITS definition of the term implement.)

> Variance: A variance exists and is defined in the legalese terminology of "obtaining a variance (exception)" to some law by bootlicking and bribing the appropriate petty bureaucrats. It is also the square of the standard deviation in the terminology of statistics. A variance is not a variation on an instruction's operation, that is unless one wished to redefine conventional usage.

I have been collecting reports from several subscribers on the Altair product and with the exception of what appear to be relatively minor technical problems, most purchasers of the system indicate satisfaction with the product and service on it.

ERRATUM:

Charles S. Lovett receives a one issue subscription extension for being the first subscriber to report an error in the ECS-7 design article of February 1975 ECS. The line from pin 2 of IC -14- which is shown connected toground should instead have been a .01 mfd capacitor to ground. (Switch S1 would have no effect if wired as drawn.)

A NOTE CONCERNING THE MOTOROLA 6800 MPU...

With this issue, I have started to make references to the M6800 MPU system, primarily because I expect it to be available to the Experimenter's Computer System market in the near future. I have been in fairly close contact with the local Motorola sales office in connection with some hardware/software design work I am currently doing, and I have indications that supplies of this product will soon be fairly widely distributed.

If you want to find out about the M6800 in detail, I wholeheartedly recommend purchase of the M6800 Microprocessor Applications Manual (approximately 700 pages 8.5 x 11 @ $25.00) and the M6800 Microprocessor Programming Manual (approximately 250 pages @ $10.00). The applications manual includes lots of useful information including interfaces (hardware and software) to floppy discs, cassette tape drives, teletype, Burroughs self-scan displays, adding machine tape printers, etc. etc. I have verbal assurances from the local Motorola sales office that these books will be sold to private individua on request. If you are interested I suggest that you look up the telephone number of the nearest office and inquire. If you have any problems, let me know and I'll try to make formal arrangements to distribute copies. These documents will set the standard for some time to come, and would easily serve as the basis of a "software engineering" course in applications.

ECS - The Monthly Magazine of Ideas for the MICROCOMPUTER EXPERIMENTER...

News & Notes to accompany Volume 1, No. 4 - April 1975. Some further midnight madness...

FLOPPY DISC DRIVE REPORT: I spent some time talking to Don Whitehead this week concerning the floppy disk purchase and its progress. Here is the latest status report on the operation:

a. There is sufficient interest to warrant going through with the purchase as originally intended ... BUT ...

b. When Don went back to the Memorex representative to make the firm commitment on an order of the drives he found several business points had changed from the time of his preliminary arrangements, to whit:

- The delivery dates are getting pushed steadily back by the manufacturer as priority is given to the larger purchasers of the unit. Current estimate is June - original was April.

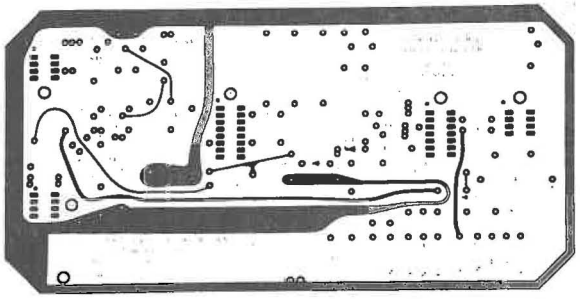
- Memorex will not allow the technical details of the interface (ie: detailed manuals) be distributed for at least six months for competitive reasons. This is despite all assurances to the contrary earlier. The demand for a non-disclosure agreement effectively rules out distribution of technical specs, thus in itself wipingout any possibility of using the Memorex drive.

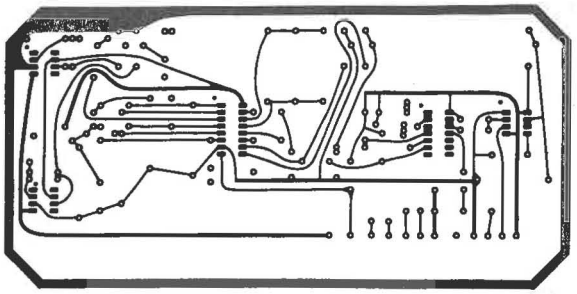
c. The individuals sending in the deposits have all had their checks returned for the time being, until a new arrangement can be made with an alternate source.

The idea and intent are not being abandoned due to this setback. Among other things, Don wants to locate the source for his own consulting software business. Between now and the next progress report, we'll both be doing a bit of research on several other options available in regard to floppy disk drives.

JIM FRY INDICATES to me in a letter that he will be repeating the memory IC offer in May of this year. Again, his address is P.O. Box 6585, Toledo, Ohio 43612 - write him for details on the 2102's selling new at approximately $5.00.

PC BOARD FOR ECS-8. After one false start with a vendor who could not deliver, the ECS-8 modem PC boards have arrived (April 15) and been shipped to all subscribers. who ordered the product. The boards have a layout looking (roughly) like this: (preliminary.)





The price is $9.00 plus postage for 4 ounces (approximate), and there is no discount applicable to this item.

TWO PUBLICATION REFERENCES WHICH MAY BE OF INTEREST TO SUBSCRIBERS:

Hal Singer (Cabrillo Computer Center, 4350 Constellation Road, Lompoc, Ca. 93436) puts out the Micro-8 newsletter, devoted to information concerning the original homebrew 8008 system of Radio Electronics summer '74.

The Computer Hobbyist, 520 Sorrel Street, Cay North Carolina 27511 is a publication for which I only have an existence proof - several new subscribers referencing a letter from Gordon French.