H. A. Cohen and R.S. Francis, *Macro-Assemblers and Macro-Based Languages in Micro-processor Software Development*, in P. Isaacson (Editor), Microprocessors and Microcomputers, IEEE Computer Society, New York, 1979, 2nd Revised Edition, pp 199-210 Reprinted from IEEE Computer, Vol 12, No 2, Feb 1979, pp 53-64

Using a conventional macro-assembler as a multi-target cross-assembler and cross-compiler, the authors developed portable algorithmic *code for microprocessor controllers.*

Macro-Assemblers and Macro-Based Languages in Microprocessor Software Development

Harvey A. Cohen*
Rhys S. Francis
La Trobe University, Australia

Although most advanced minis and mainframes boast an assembler with advanced macro facilities, only rarely is significant use made of such facilities. Yet, as we explain in some detail here, a macro-assembler is (almost) a universal X-assembler. Moreover, through the development of an assembly-time stack for message passing between macros (at macro-expansion time), a macro-assembler can serve as a universal X-compiler for higher-level languages.

We exploited these macro-assembler capabilities in constructing a multi-target microprocessor development system resident on a DEC-10 mainframe. Developed for the Oznaki Educational Project, this system has as prime utilities CROZZ and HELL. The macro-cross-assembler CROZZ supports the E080, 6800, SC/MP, and 6502 processors, and can be readily extended by the user to other target processors. HELL—a "highly extensible luverly language"---is a special-purpose macro-based language, with an Algol flavor. It yields code for **the** 8080 and 6800 processors, as well as the DEC-10 mainframe.

In order to meet in advance the challenges of minor hardware alterations ("update") in a particular microcomputer, and the challenge of transporting software from one microcomputer to another, we have developed a code development strategy called HELP. HELP involves a code partition scheme, with well-defined links between what we term *algorithmic* and *interfacial* code.

Background

The continuing emergence of microprocessors and bit--slice processors presents a variety of challenges to those responsible for software development. One of the most basic is simply to produce the utilities and programming system in which to develop application programs for a range—possibly even a great range—of different processors. While awaiting the development and/or availability of basic utilities, many programmers have been obliged to resort to archaic hand assembly and direct machine loading. A few companies and other organizations have attempted to minimize this by restraining system designers to a particular microprocessor, invariably one supported by a custom development system. One can safely predict that when new processors emerge (with specifically desirable capabilities), efforts at standardization on a particular processor will fail. And where a custom development system is used, the availability of basic utilities for any new processor will depend on the marketing strategy of the manufacturer.

To overcome these difficulties we advocate the use of an advanced mini or mainframe already familiar to the programming team as the host machine for a multi-target development system. This approach makes the host machine's existing editing, file management, back-up, and archival systems available. In order to follow this approach, cross-assemblers for many target processors must be available. Later we will show how the macro-assembler of the host machine can serve as a universal X-assembler.

Microprocessors pose a further challenge—development of cost-effective strategies for the implementation of higher-level languages for a multiplicity of processors. It is well-known from tradi. lanai "software engineering" that programming a higher-level language reduces both the programmer's effort and the number of errors, yields more intelligible programs, and provides the opportunity for developing processor-independent code.

*Currently on study leave at the Division for Study and Research in Education, MIT, Cambridge. Mass. 02139

However, traditional implementations of high-level languages require massive effort in the first place. Moreover, the effort required to update such languages for processor and architecture variations becomes ridiculous for the purposes of control-type applications involving programs of only modest size. For such control-type applications, we advocate the development of a problem-oriented, macro-based language for each family of applications. Each language will consist of the same core of control structures and primitive data statements, to which are added only those constructs relevant to the particular problem area. The starting point for such development is the use of a macro-assembler as a universal X-compiler. As the macro-assembler already supplies all the necessary parsing and bookkeeping support, programmers unversed in compiler or systems programming can implement languages in minimal time. However, since none of the texts on macro-processors present even rudimentary ideas on how to use a macro-assembler, we present an introductory discussion here.

Programming in a high-level language immediately leads to the possibility of producing portable software. It is clear that if software is to be portable, it must be written in a source language that does not contain any references to processor-dependent information. However, the programmer of microprocessor-based systems is commonly concerned not only with what is to happen but also with the hardware dependencies of how it is to happen. Consequently, a myriad of hardware details is scattered throughout the typical application program. In another paper' we proposed that effective code production for microprocessor-based systems requires the separation of code into interfacial and algorithmic parts (briefly mentioned above), with specialized linking code forming the bridge between the two. The algorithmic software, having had hardware dependencies removed, lends itself to implementation in .a higher-level language. The algorithmic code can be developed by "pure" programmers, leaving the development of interfacial code to those with hardware comprehension. The interfacial code must be rewritten for each hardware configuration. But the algorithmic code is in principle transportable.

The ideas we describe here have been applied to software development for the Oznaki Project. In Ozpaid., microprocessor technology has. been used to construct interactive "models" of mathematical systems. In these computer models/the student programs the activities of robots called "Naleis," which create patterns and designs, thread mazes, and chase targets about a TV screen. The tiny languages in which the student programs these t-Takis are intended for implementation on exceedingly low-cost controllers.* For the moment, "personal computers" based on 8080 and 6800 processors are used in preliminary educational studies. The project has access to a DECsystem-10 mainframe, used as the host machine for our development system. The basis for our implementation was the macro-cross-assembler package CROZZ Zoi various microprocessors, Inherent in the project is a. special interest in microprocessor code production, update, and transport problems. The "highly extensible luverly language" — HELL, based on macros for the DEC-10 Macro-10 macro-assembler, has been developed and used as the prime implementation language. This language *cBn* be processed to produce code for the DEC-10 itself (to provide a variety of emulation we call high-level emulation) as well as for the 6800 and 8080 processors.

*With specifications similar to that of the Texas Instruments "Speak and Spell" —see R. Wiggins and L. Brantingham, "Three-Chip System Synthesizes Human Speech," *Electronics,* Vol. 51, No. 18, Aug. 31, 1978, pp. 109-116.

Interfacial and algorithmic code

We begin by discussing the code separation required prior to the implementation of a high-level language software development system. To make any headway on the problems of code production and update it is necessary to examine the nature of the software changes induced by hardware configuration changes. Major portions of any well-structured code are unaltered by such changes. To eliminate once and for all the sheer messiness of having to make innumerable tiny alterations in what had been functional code, we need to erect a barrier within the sof t-ware. Outside this barrier lies software not affected by hardware configuration changes. This software is essentially configuration-free; we call it algorithmic software. Within the barrier lies the interfacial software, including such items as input, output, and timing routines, special execution mode routines, and interrupt handlers. Our development strategy requires that properly constructed programs should never directly penetrate the barrier between algorithmic and interfacial code. Instead, all communication between the two must be via specially provided bridges or links.

This modularity can be extended further to solve the update problem for a system where distinct function code is placed in different ROMs. If one ROM is altered for whatever reason, one wishes to avoid having to alter other ROMs. This can be achieved if each ROM has an associated link wall to provide definite bridges into the code contained within it. Thus a particular ROM can be replaced without affecting the other ROMs, provided that the link wall is maintained.

These bridge specifications can be viewed as a shift in the normal division between hardware and software. Hardware designers have traditionally seen their role as being limited to the production of functional electronics. They then present software writers with a precise hardware specification replete with bit-specific details. As a result, changes in hardware require an extensive software amendment effort with it impractical to update existing units in the field. And yet there will be innumerable possibilities for economies and improvements, resulting from the variety of combinations of logical functions available on single chips, together with those which will

H. A. Cohen and R.S. Francis, *Macro-Assemblers and Macro-Based Languages in Micro-processor Software Development*, in P. Isaacson (Editor), Microprocessors and Microcomputers, IEEE Computer Society, New York, 1979, 2nd Revised Edition, pp 199-210 Reprinted from IEEE Computer, Vol 12, No 2, Feb 1979, pp 53-64

become available within the life of a product. With the traditional demarcation between hardware and software if, is difficult to take advantage of these possibilities. However, by creating a demarcation between the algorithmic and interfacial code an effective simplification of the update problem can be obtained For example, the first version of a product might use software to perform serial I/O whereas a later version might use an enhanced chip to perform the aerial I/O in hardware. In this case, new interfacial code would have to be produced, but existing algorithmic software would function on the new hardware without alteration, More importantly, any new algorithms programmed for the device would function on both the nee e !.trait and those already produced.

Another way of looking at this can be gained from a study of any piece of applications software. All such programs have as their basis a conceptual hierarchy of operations. The programmer uses the control structures of the particular programming language to organize these operations to provide the desired performance. When considering assembly language programming, the lowest=level operations are the machine instructions provided by the particular processor (ignoring the possibility of microprogramming). The next highest level of operations consists of those sequences of machine code required to control or interact with the various devices connected to the processor, The next level comprises those routines providing the logical operation of these devices. Examples could be routines to "output a character to a video terminal" or "wait for the next character from the keyboard." it is important to appreciate this change to a logical operation level. It is irrelevant to the calling code whether the video terminal is memory-mapped or connected by a serial data link, or whether the keyboard is buffered or unbuffered. It is these details that make up the code bodies of the operations at this level. All more complex operations, constructed in terms of those primitive logical operations, must themselves be logical in nature and therefore hardware-independent, Following the conception and specification of these primitive logical operations, it, is clear that the code implementing them in terms of the available hardware belongs to the interfacial category, .while the code written *in terms of the primitive operations themselves* belongs to the algorithmic category.

Thus the first problem in system development is deciding the sorts of primitive logical operations needed for the intended application domain. This is a very important and difficult procedure and should be approached after the program family outline has been completely clarified. The next step is specification of the memory space partitioning between the interfacial and algorithmic software, and the details of the bridges between them. These bridge specifications require careful construction.[2] Indeed, to simplify update problems we must employ elaborate strategies to give these bridges a physical existence ire memory (as f red location address tables, for example). As a result of making these decisions the software problem can be neatly divided into two parts. The details of the interfacial software are irrelevant to algorithmic programmers, who only need to know the bridge specifications. Likewise interfacial code programmers are only concerned with providing the functions inherent in the bridge specifications, in terms of low-level hardware-oriented code. It is here that the hardware-to-software boundary can be specified and re-specified, as such decisions concern only interfacial code programmers and hardware designers.

Macros

Macros are traditionally introduced as a means of extending the instruction set of an assembler. To quote the now classic example (wherein the expansion is for the M6800 processor) the definition

```
DEFINE      SUMS P1,P2,P3) <
            LDA          A,PˆI
            ADD          A,P2
            STA          A,P3>
```

implies that the statement

```
            SUMS (100, NUMB, RESULT)
```

occurring in the assembly code will generate the machine code corresponding to

```
            LDA          A,100
            ADD          A,NUMB
            STA          A,RESULT
```

This adds (modulo 256) the contents of the bytes addressed by the first two parameters, in this case 100 and NUMB, and stores the sum in the byte addressed by the third parameter—in this case RESULT. As all three parameters are assumed to be addresses, both NUMB and RESULT would have to appear as labels somewhere in the assembly.

Clearly the macro definition has added an entirely new and higher-level construct to the assembly language. Moreover, the cell or invocation of sums contains no processor-dependent information. By substituting a different definition of SUMS, the same source statement could be made to produce valid code for any other processor. For example, for the IN8080 the definition of SUMS would he

```
DEFINE      SUM8(P1,P2,P3)<
            LDA          P1
            LXI          H,P2
            ADA          M
            STA          P3>
```

or more interestingly, for the DECsysterri-10

```
DEFINE      SUM8(P1,P2,P3) <
            MOVE         0,P1
            ADD          0,P2
            ANDI         0.fB11111111
            MOVEM        0,P3>
```

Note that for the DEC-10 we have arbitrarily chosen to use register zero as an accumulator, and we have taken care to appropriately model the 8-bit nature of the operation implemented.

From this it is evident that if the basic operations required for a group of algorithms are known, development of machine-independent software becomes possible by defining suitable macros for these operations. There are four points we should note about the above examples. We will discuss each in detail later.

- In order to expand the resulting source software for a given processor, it is necessary to have a macro-cross-assembler for that processor running on the host machine. Further, to allow for portability of algorithmic software between different processors, it is mandatory that each cross-assembler support identical assernbly-time and macro facilities.

- The simple example above might leave the impression that the resultant source code will look like a list of Fortran-type subroutine calls. This is certainly not the case, as it is possible to implement structuring constructs such as

```
        IF (RELATION) THEN
        ........
        ELSE
        ........
        ENDIF
and
        LOOP
        ..........
        EXITIF (RELATION)
        ........
        ENDLOOP
```

  using the facilities of a normal macro-assembler, despite its usually restricted format for parameter parsing.

- The modularity of macros allows new statements, control structures, or data types to be easily added to the implemented language. It also permits refinement of existing -macros, and thus allows improvements in the efficiency of the generated code (to be made in parallel with the development of the applications software).

- By maintaining a suitable library of macros for host machine applications, algorithms can be executed and tested on the host machine *using* its existing debugging facilities.

Macro-assemblers as universal X-assemblers

The basic tool in a microprocessor code development system is the cross-assembler. To implement systems as described in this paper requires the production of cross-assemblers containing sophisticated macro facilities. Fortunately, the means of achieving this is readily available in the form of the host machine's macro-assembler. In the past it has been little recognized that adding macro facilities to an assembler converts it into a universal assembler. It is universal in that the macro facilities provide the ability to assemble programs for processors other than the one for which the base assembler was intended. To clarify this, consider the output listing from virtually any assembler. It consists of some numbers, usually on the left of the page, and some symbols, usually on the right. In an ordinary assembler, the process of assembling embodies preset rules for translating the symbolic code (source code) into the numeric code (object or machine code). The addition of macro facilities to the assembler allows the user to define new rules for specifying the relationship between the symbols and the numbers. Given that the machine code for any processor can be represented by numbers, and given that these numbers can be made to appear in the output listing as a result of user-specifled symbols appearing in the source text, it is clear that a macro-assembler is a universal assembler.

We demonstrate the use of a macro-assembler as a universal assembler by reference to Macro-10, **the** resident assembler for the DEC-10 mainframe. In the following source file macros are defined to have **the** names of certain IN8080 mnemonics. The pseudo-op EXP appearing in this listing assembles a 36-bit word equal to the following parameter:

```
TITLE           EXAMPLE

                OPCODE DEFINITION

DEFINE          LDA(ADDR)<
                EXP         032
                EXP         ADDR&377
                EX?         ADDR/400
DEFINE
                OUT(PORTK
                EXP         323
                EXP         PORT&377

DEFINE          HALT<
                EXP         166


                CROSS ASSEMBLY

SALL
                LOC         100
                OPORT= 5
START:          WA          CHAR1
                OUT         OPORT
                LISA        CHAR2
                OUT         OPORT
                HALT
CHART:          EXP         "X"
CIIAR2:         EXP
                END
```

This file consists of two parts. The first contains **the** macros which define the target processor's opcodes, which here comprise three instructions of the IN8080 instruction set. The second part appears as the normal input would to an IN8080 assembler. Submitting this file to Macro-10 produces the listing at top opposite (note that numbers are in octal).

The "object code" produced by Macro-10 appears on the left-hand side of the listing and consists of two parts, the value of the PC or word address, followed the value of the object words. Macro-10 considers these assembled words to be 36 bits each and prints them in DEC-10 half-word format. For our purpose the fact that Macro-10 treats these words as 36 bits is irrelevant. **Instead** we see on the listing the PC value corresponding to the address in the microprocessor's memory, and next to this the specification of the value of the byte at that address. This listing represents a valid cross-assembly of the 1N8080 symbolic code, given in the source file. Thus Macro-10, or any other macro-assembler, can be used as a universal assembler.

However, the listings generated by Macro-10 are in a format designed to enhance intelligibility of DEC-10 machine code rather than 8-bit microprocessor code. In order to construct a more useful cross-assembly utility, the following facilities have to be provided;

- hex representation of numbers in the input file;
- hex representation in output of PC and "assembled code";
- error flagging of host machine operators and other predefined symbols meaningless to the target processors if they appeared in the source text without having been assigned new values by the user);
- suppression of undesirable pseudo-ops irrelevant to the target processors;
- ability to output error messages onto the listing from the opcode-defining macros;
- generation of appropriately formatted load files; and
- ability to load libraries of opcode-defining macros, rather than having to insert them at the start of each source program.

Only the last of these facilities is readily available in Macro-10. In order to implement the CROZZ macro-assembler package for the DEC-10, we wrote a pre-editor and a post-editor. The pre-editor simply scans the input file looking for numbers not ¯flagged for radix and replaces them with the equivalent octal representation. The post-editor is- slightly more complicated. As well as scanning the output file converting octal numbers back into hex format, it tidies up the object code presentation. It also provides routines for generating appropriately formatted load files from the data edited on the list file, as Welt as routines for allowing multiple object bytes to appear on the same line. By implementing these· Editors in DEC-10 assembly language and taking advantage of inter-task communication facilities, we attained a tidy and efficient cross-assembly package support-

```
EXAMPLE       MACRO %5012721 13:09 1¯6-FEB-78 PAGE1
OUTPUT        MAC 16-FEB-78 13:08
                                            TITLE EXAMPLE

                                       ; ;            OPCODE DEFINITION

                                           DEFINE LDA(ADDR) <
                                                  EXP        032
                                                  EXP        ADDR&377
                                                  EXP        ADDR1400
                                           >

                                            DEFINE OUT1PORT) <
                                                  EXP        323
                                                  EXP      PORT&3777
                                            >
                                            DEFINE HALT<
                                           EXP                166
                                           >

                                                                 .

 000100                                     LOC      100
                    000005                  OPORT= 5
 000100   000000    000032     START:       LDA      CHARI
 000101   000000    000113
 030102   000000    000000
 000103   000000    000323                  OUT      OPORT
 000104   000000    000005
 000105   000000    000032                  WA       CHAR2
 000106   000000    000114
 000107   000000    000000
 000110   000000    000323                  OUT      OPORT.
 000111   000000    000005
 000112   000000    000166                  HALT
 000113   000000    000130     CHAR':       EXP      "X"
 000114   000000    000101     CHAR2:       EXP
                                            END
NO ERRORS DETECTED
PROGRAM BREAK IS 000000
CPU TIME USED 00:00.224
3K CORE USED
```

ing the 6800, 8080, 6502, and SC/MP. In CROZZ the actual target processor information is contained in macro libraries for each processor, so that extension to other processors entails only writing such a macro library file, To demonstrate the effectiveness of the editing we present a CROZZ output listing for the previous example:

```
      OZNAKI PROJECT CROSS ASSEMBLER OUTPUT
      EXAMPLE
                     OUTPUT          LST       16-FEB-78 15:15
                                     TITLE      EXAMPLE

                                               OPCODE DEFINITION

                                     SEARCH X8080

                                     ;;         CROSS ASSEMBLY

                                     SALL

 0040                                ORG       40
              5                      OPORT=5
 0040                 3A4000         START:    LDA       CHAR1
 0043                   0305                   OUT       OPORT
 0045                 3A4C00                   LDA       CHAR2
 0043                   D305                   OUT       OPORT
004A                      76                   HALT
00413                     58         CHAR1:    EXP       "X"
 004C                   . 41         CHAR2:    EXP
                                               END
NO ERRORS DETECTED
PROGRAM BREAK IS 000000
Al3SLUTE BREAK IS 000161
CPU TIME USED 00:00.226
3K CORE USED
```

The technique detailed above—for using a macro-assembler as a universal X-assembler—is not to our knowledge described anywhere in the literature. We discovered this method in April, 1976. It appears to have been discovered (or perhaps rediscovered) elsewhere at about the same time—thereby proving that necessity is the mother of invention. In fact the technique might well have been discovered at any time since the first macro-assemblers were implemented in the early '60s. It is surprising that the early workers Halpern[3] and Ferguson[4] do not mention the ability of a macro-assembler to function (albeit crudely) as a "meta-assembler" in which the user can specify mnemonics and opcodes.

For completeness we mention another technique for cross-assembler construction described in a number of recent papers.[5] In this approach the user examines the structure of the coding of an existing assembler, and alters the table of mnemonics, the **factors** related to word size, and the address arithmetic. Of course the better assemblers with enhanced macro facilities are not in the public domain, so only in special circumstances can this method yield a cross-assembler with advanced facilities and efficient execution.

Jenkins and Tyers[6,7] have assessed a number **of** proprietary cross-assemblers for various processors. These assemblers, written in Fortran, offer the user the most rudimentary facilities and yet compile code 10 to 40 times slower than CROZZ. This gross inefficiency has generally been compounded by the use **of** table look-up to convert from ASCII input and output to an internal EBCD representation.

In comparison, by using the macro technique described above, one quickly obtains a well-tested, sophisticated cross-macro-assembler. It is well-tested in that once the opcode-defining macros are correct, the cross-assembler will have only those bugs which exist in the host assembler. Where the host machine has been in service for some years, such bugs are (generally) insignificant. It is "sophisticated" in that such a cross-assembler automatically possesses all the assembly-time macro features of the host assembler. There only remains the question of implementation time. While the library of macros for a particular processor can initially be written in a day, our experience suggests that to add a tested, debugged, and documented cross-assembler to the CROZZ system requires approximately two man-weeks, nearly all of which is spent in the testing and documentation. The pre- and post-editors can, in the first instance, be as simple as possible, be written in any convenient language, and represent a few days work at most. The emphasis here is to get a development system available for the production of microprocessor code and then to refine it where it is necessary or desirable. The editors used in CROZZ have undergone several such developments, with the latest version written in DEC-10 assembly language for maximum speed. What is important is that these developments were made as and when time became available, and in no way interfered with the availability and use of the system.

Macro-assemblers as universal Xcompilers

The previous section demonstrated that a macro-assembler *can* be used as the basis of a Microprocessor code development system. Additional advantages accrue from the fact that the resultant cross-assemblers contain sophisticated assembly-time *macro* facilities such as conditional assembly pseudo--ops. Using these it is possible to define code-structuring macros that in effect allow code to be written in a special-purpose, problem-oriented, high-level language. As an example we demonstrate the construction of (run-time) conditionals. Consider an implementation of the Algol statement:

```
IF JILL = JOE THEN
           BEGIN FLIP END
ELSE
           BEGIN FLOP, END;
```

in terms of the macros IFEQ, THEN, ELSE, and ENDIF

```
1FEQ              JILL, JOE
THEN
                 FLIP
ELSE              FLOP

ENDIF
```

It is helpful to compare the desired source text with the Intel 8080 assembly language code required to implement its function:

```
        LDA        JILL            IFEQ    JILL,JOE

        LXI        H,JOE
        CMP        M
        JNZ        LABEL(1)
                                   THEN
        FLIP                               FLIP

        JMP        LABEL12)
LABEL(1):
        FLOP                       ELSE    FLOP
LABELI2):                          ENDIF
```

Clearly two labels, which we have signified by LABEL 11 and LABEL (2), have to be created by CROZZ-8080 to mark the beginning and the end of the "flop" coding. For the moment we overlook the possibility of nested conditionals, and write macro definitions that would correctly assemble the left-hand side above:

```
DEFINE IFEQ(UNK
          N=N+1
          LDA            U
          LXI            H,V
          CMP            M
          JNZ             LABI .(N)>

DEFINE THEN< >          ;THE IS A NULL OP

D EFINE ELSECVK
          LABELCN1=.
          N=N+1
          JMP            LABI LAN)
          Y>

D:FINE ENDIF<
          LABEL(N)=.
```

In these definitions the period "." stands for the present value of the program counter. To make the above definitions valid, at the start of the program the index N would have to be initialized to zero. After this, the vector of created labels, LABEL W, LABEL (2), LABEL 13), - would be augmented each time IFEQ was called. Actually, CROZZ does not have available the notation we are using of a vector of labels. Instead it is necessary to define a macro which accepts the index as an immediate evaluation parameter, and using character concatenation, generates a unique symbol. The 'most important point is the maintenance of a one-to-one correspondence between the index values and the generated symbols.

The definitions given above will work even if the ELSE is present for any number of consecutive conditional statements but fails for nested conditionals. For example, in the following construction:

```
IFEQ         A,B
THEN         FLIP
ELSE         IFEQ         C,D
THEN         FLOP
             ENDIF
ENDIF
```

the final label inserted by the outer ENDIF would have the same name as the label inserted by the inner ENDIP. However, nesting of IF „ THEN . ELSE . . . constructs satisfies the usual principles of block-structuring, so there is a way around this which is simple in conception but requires careful coding to implement. Suppose the macro SPUSI-I places the MP. rent value of the variable IFNUMBER on an assembly-time stack. And suppose further the macro $POP sets N equal to the top of the stack when it pops the stack. Using these two new macro-pseudo-ops, the prey ions definition of IFEQ can be refined as follows:

```
DEFINE IFEQ(U,V)<
        IFNUMBER=IFNUMBER+ 2
        $PUSH           ;SAVE PREVIOUS N
                        VALUE
        N =IFNUMBE R
        LDA
        LXI
        CMP
        JNZ      LABEL(N)>

DEFINE THEN< > ; ;THEN IS A MACRO NO-OP

DEFINE ELSE(Y)<
        LABEL(N)=.
        N=N+ 1
        JMP      LABEL(N)
        Y>

DEFINE ENDIF<
        LABEL(N)=.
        SPOP            :RESTORE N>
```

Using these macro definitions, the vector LABEL (N) will have "holes" corresponding to the missing ELSE, which is of no consequence. The generalization of IFEQ by an IF macro, that takes as one of its arguments such character strings as EQ for equal, LE for less than, **GE** for greater than, and the like, is very straightforward.

```
DEFINE    IF(ADDR1,REL,ADDR2)<
          IFNUMBER=IFNUMBER+2
          $PUSH
          STEMP=1
          N=IFNUMBER
          IFIDN<REL><EQ>,<
             LDA        ADDR1
             LXI        H,ADDR2
             CMP
             JNZ        LABEL(N)
             $TEMP=0
          IFIDN<REL>-(GE>,<
             LDA        ADDR2
             LXI        FLADDR1
             CMP
             JM         LABELIN)
              TEMP-Q >
          IFIDN<REL><LE>,<
             LDA        ADDR1
             LXI        H,ADDR2
             CMP
             JM         LABEL(N)
             STEM P =0 >
1FN $TEMP, <PRINTX ERROR IN IF> >
```

This definition includes simple error reporting code involving STEM?. In CROZZ macros detecting errors set particular bits of the DEC-10 word, and the output listing is flagged for error type.

Note how the assembly-time stack serves to pass messages from one macro to another. Such a use of an assembly-time stack was made by Herman-Giddens, et al,' in adding block structures to POI'-11 assembly language. In the macro definitions presented above, the parameters are required to be (16-bit) addresses. While this is adequate for simple purposes, a significant high-level language must permit expressions in lieu of simpler parameters.

In HELL, the macro-based implementation larigua.ge used in the Oznaki Project, generalized expressions are parsed via a carefully arranged set of recursive definitions that also exploit the assembly-time stack. This expression parser is effectively as complex as those in other higher-level languages, but otherwise, Macro-10 supplies all other bookkeeping support so that the implementation labor is small. The detailed features of HELL are listed on the next page, together with a demonstration program written in HELL. This demonstration program controls a rudimentary robot. The source listing has been used to generate code to control the robot from the mainframe DEC-10 and from microcomputers based on both the 6800 and 8030 MPUs.

There have been many papers published concerning the use of macro-processing for language implementation support, most recently Furgerson and Gibbons,[9] Rechenberg,[1o] and Tanenbaum." Most of these have described the use of a general-purpose macro-processor with features and refinements not available in macro-assemblers. However, we have found no references in the literature to the use of a macro-assembler as a universal Assembler, or more importantly, as a cross-compiler supporting the development of transportable code. One notable feature possessed by powerful macro-processors is that macro definitions include specification of

Implementation language macros

The macro-based implementation language HELL has the following macros for use in application programs. Note that the original versions of the Oznaki Project's software were written in DEC-10 Algol60. with patched-in machine routines. In order to simplify the rewriting of this ere-existing code. HELL acquired perforce an Algol-like appearance. The list below does not include the macros for expression parsing, with names commencing with "%%".

Statement Statement	Description
Code segmentation	
DATA NUMBER	Locate following code into RAM memory and provide the specified number of words of memory for the program execution stack.
CONSTANTS	Locate following code into ROM memory.
PROGRAM TITLE	Specify the title for the listing, make the current address the program start address and insert all necessary initialization code.
FINIS	Specify the end of the program and append those library routines required for the particular program.
Control Structures	
IF (relation)	Generate code to evaluate the indicated relation and conditionally jump to the address of the appropriate ELSE or ENDIF when the relation is false.
ELSZ	insert a jump to the appropriate ENDIF and supply the label used for the jump in the previous IF.
ENDIF	Insert the label used in the preceding IF or ELSE. -
LOOP	Define a label which will be used in the lump statement inserted into the code by a following ENDLOOP.
EXITIF (relation)	Generate code to evaluate the indicated relation and to conditionally jump to the appropriate ENDLOOP when the relation is true.
ENDLOOP	Insert a jump to the appropriate previous LOOP and define the label referenced in the previous EXITIF.
PROCEDURE name	Define a macro with the specified name which performs a subroutine jump to the current address, (the address of the line on which the procedure statement occurs).
RETURN	Generate. code to perform a subroutine return.
GOTO label	Generate code to perform a jump to the specified address.
SWITCH VAR,TOP, BOT,<label list >	Generate code to check that the value of the given variable between the limits `TOP' and 'BOT' and if so to transfer to the appropriate label from the list.
Data statements	
INTEGER <list of names >	Allocate consecutive bytes of memory to store single byte integer variables referenced by the given names.
STREAM name, length	Allocate a block of memory of the specified length, addressable through the stream manipulation commands, using the given name (effectively **a** string of single-byte integers).
ASSIGN name,<expr >	Generate code to evaluate the specified expression and store the result in the named integer variable.
SELECT name	Generate code to select the named stream for subsequent manipulation.
BEGINSTREAM	Generate code to set the current streams pointer to the start of the stream.
INSYMBOL(variable)	Generate code to obtain the byte pointed to by the current streams pointer. increment :he pointer and store the obtained byte in the named variable.
OUTSYMBOL(expr)	Generate code to evaluate the specified expression and store the single-byte result in the ariOre3s pointed to by the current streams pointer, and then increment the pointer.
SKIPSYMBOL(ex.pr)	Generate code to evaluate the specified expression and to increment the current streams pointer by this value.
BACKSYMBOL(expr)	Generate code to evaluate the spe--cified expression and to decrement the current streams pointer by this

Application program demonstrating HELL **syntax**

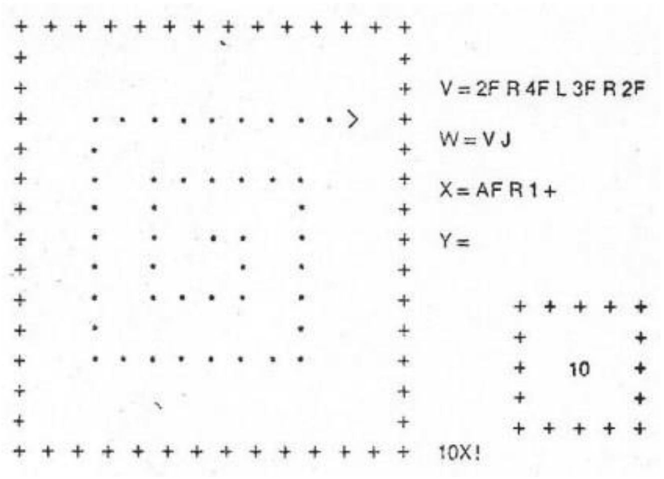
```
TITLE OZJUNIOR
:COMMENTS PRECEDED BY ";" ON SAME LINE
SEARCH X8Gt30            ; Specifies processor
SEARCH WIZBOX           ; Specifies    MACHINE
                          ;Configuration
                          Linkwall, Bridges,
                             and library
                             macros
```

H. A. Cohen and R.S. Francis, *Macro-Assemblers and Macro-Based Languages in Micro-processor Software Development*, in P. Isaacson (Editor), Microprocessors and Microcomputers, IEEE Computer Society, New York, 1979, 2nd Revised Edition. pp 199-210 Reprinted from IEEE Computer. Vol 12. No 2. Feb 1979. pp 53-64

```
        :ASSEMBLY TIME VARIABLES
        NULL-0
        LINEFEED = OA
        CARETN 0D

DATA (10)                       ;STACK OF LENGTH 10
                 :PLACES CODE FOLLOWING IN RAM

        VARIABLE <CMD. NUMF3, MODE, MTRCMD >
                 :DECLARATION OF SINGLE-BYTE VARIABLES

CONSTANTS        :LOCATES SUBSEOUENT CODE IN ROM

 PROCEDURENEWLINE
        :USES LIBRARY ROUTINE OUTCHAP.
        OUTCHAR(<VALUE LiNEFEED>i
        OUTCHARi<VALUE.CARETN>)
                <VALIUE.NULL%l
                     E.NULL>
        OUTCHARKVALUE,NULI.>)
        RETURN

 PROGRAM WIZ:
        ;INITIALIZATION ROUTINES INSERTED HERE BY HELL

 NEWCMD:         ASSIGN MODE, <VALUE,0>
        ASSIGN    NUMB,<VALUE.1>
 INCMD:          INCHA.R(CM0)
        OUTCHACMD)
        F        WMO.E0.<VP.LUE.¨H">)
                ASSIGN M3K1E.<1011,MODE,<VALUE2.>>
                GOTO INCMD
        END:F
        IF       f_CY'D.E0.<VALUE.¨A¨>)
                ASSIGN MODE.<1CR.MODE,<VALIIE.1>>
                GOTO
        ENDIF
        IF      1.<CMD.GE,<VALUE.'*0¨>>.AND,<CMD,LE.<VALUE,"9">>)
                ASSIGN NUMB,<SUB.CMD,<VALLIE,"0">>
                GOTO INCMD
        ENOIF
        IF       (CMO,E0,<VALUF,"F">)
                 ASSIGN MTRCMD.< VALUE,60>
        ELSE
                 IF      (CNID,E0.<VALLIE."B">)
                         ASSIGN MTRCMD.<VALLIE.6C>
                 ELSE
                     IF (CMD,E0<VALUE,"R">)
                         ASSIGN MIRCMD,<VALUE.68>
                     ELSE
                         IF (CMD.E0.,<VALUE,"L¨>)
                             ASSIGN MTRCMD,<VALUE,64>
                         ELSE
                             IF (CMD,E0.<VA1.UE."3¨>)
                                 ASSIGN MTP.CMD,<VALUE,70>
                             ELSE
                                 GOTO NEWCMD
                             ENDIF
                         ENDIF
                     ENDIF
                ENDIF
          ENDIF
 MT ROUT.        ASSIGN MTPCIii1D,<A0D.MTROMD.MOD✓
        NEWLiNE
        LOOP
        EXITIF IF NUMB  LE<VALUE.0>)
                ASSIGN  NUMB <SUE,NUMS,<:VALLJE,1>>
                ASSIGNN MOCE.VALLIE,1:7'
                LOOP
                    CUTONAMM¨fRCC!0)
                EXITtF           FO<VALUEA
                    PAUSP,<VALtiE.40>)
                           MODE,<ADD,MODE,<V.4LUE,1>>
                ENDLOOP
                LOOP
                EXITIF (MODE.EG,<VALUE,O>)
                    • PAUSE(<VALUE,60>)
                      ASSIGN MODE,<SUB,MODE,<VALUE,1>>
                ENDLCIOP
                NEWLINE
        ENDLOOR
        GOTO        NEWCM 0

        FINIS
```

The above program accepts and decodes character commands from a keyboard and outputs control commands to a small robot. Keyboard commands are" F" for forward, "B" for back, "R" for right turn, "L" for left turn, "6" for sit (motors off for specified duration). and "0" to "9" specifying the number c repetitions. There are also the "mode" modifiers "A" for turning on the robot's lights and "H" for honking a horn. A typical keyboard command steam would be 2HF 3ARBHAS, which would be translated by the Ozjunior program so that the robot takes 2 forward steps while honking, then turns through 3 unit turns with lights on. etc. The Ozjunior program is a cut down version of the program *Oz.* The complete Oz program includes facilities for the student user to define and call by name macros of robot and display commands. Note that the above program, written in a 1976 version of HELL. required actual numbers to be preceded by the symbol "VALUE-". This device, reminiscent of the Lisp quote, was required because of a bug in the implementation of the Macro-10 pseudo-op

Another example of Oznaki software is provided by the *game Wham, in which the user directs a "robot" called a "Naki" around a TV screen. The Naki has the appearance ^,>,V or < depending on its heading, and deposits asterisks us it moves. Wham combines a discrete geometry with calculator maths.



The above printout of a TV screen for Wham was taken after the user had defined V, W. and X macros, and used the X macro to draw the asterisk "squiral." (The parameter A is the number in the small square accumulator on the right.) The HELL program for Wham assembled in approximately 3K byes for 8080 and 68OO processors, so Wham is indeed *tiny* To achieve portability, the action of putting a character on a screen was used as a logical operation whether achieved by placing a location in memory for memory-mapped video or by outputting an appropriate sequence of characters to an external terminal. A copy of the screen was stored in data streams in approximately 5K byte of memory leading to essential redundancy in the memory-mapped video architecture.

parameter scanning. This greatly simplifies the implementation of higher-level languages, as instanced by *the* possibility of defining a single macro-called IF. This macro, when encountered, scans the source text looking for the symbol THEN, which delimits the relation, and then continues the scan to find the appropriate ELSE and ENDIF symbols. Thus the first parameter of the IF macro is the relation, the second parameter contains the statements to be executed when the relation is true, and the optional third parameter contains the statements to be executed when the relation is false. Using these parameters the IF macro cari immediately output code appropriate to the complete conditional statement. A macro-assembler usually does not have such parameter parsing facilities. Instead it is necessary to define separate macros as described and provide an assembly-time stack for communication. This may not necessarily be a disadvantage when it comes to compile-time efficiency. The Oznaki Project programs using HELL assembled at eight lines per second, which compares well with Tanenbaum's one-to--two lines per second using a general-purpose macro-processor. As he reported, most of the processing time is consumed analyzing expressions. We found the assembly speed to be critically related to the efficiency of the expression macros; we expended considerable effort in determining their optimum construction.

Macro modularity

One of the recognized advantages of a macro-implemented language is its resultant modularity. The macros defined to implement a given source language feature are in general completely independent of other macros defined to implement other language features. Adding a new statement to the language only requires the definition of an appropriate macro, assuming of course that this new feature is not meant to interact with existing features. A corollary of this is that continuing refinement of existing macros can proceed in parallel with program development. Refinement here refers to improvements in the efficiency of the generated code, as measured by code size or execution speed. Consider the following improved I N8080 definition of the SUM8 **macro mentioned above:**

```
DEFINE SIJM$P1,P2,P3K
           !FILM < P1 >< P3>,‹
                   LDA        P2
                   1,X1
                   ADA
                   MOV                  >
           IFDIF < P1 >-< PS >,<
                   IFIDN < P2 r

                             LXI          H,P3
                             ADA
                             MOV          M,A
                   IFDIF < P2 :><", P3
                             LDA          Cl
                             LXI          H,P2
                             ADA
                             STA           P3 > >>
```

This new definition generates only 5 bytes of code to perform the sum operation, whenever one of the first two addresses is the same as the result address. In other cases it generates 10 bytes of code. The important point about this example is the fact that the improvement can be conceived of and incorporated into the system at any time. It is not necessary to produce the best possible macro implementations at first. Instead, the quick production of a working library of macros performing the specified functions can be the initial objective. This then allows the development and testing of the applications software to begin. Once this has been achieved work can proceed with refining the macro implementations as a separate project. As each improvement is made the old macro definition is simply replaced by the new, after which all subsequent compilations will incorporate the enhancement.

Another benefit of this language modularity is **a** clear correspondence between the source and object code. In order to understand the code generated by a source statement only those relevant macros need be considered. In fact, the identification of source code with object code is contained precisely within the definition of the macros concerned. We feel that within the framework of producing code for micro-processor controllers from high-level languages, it is advantageous for programmers to have a clear understanding of the object code produced by each source statement. This understanding is strengthened by the tight binding of source and object code on the listings generated by the macro-assembler.

The disadvantage of strict modularity in the actions of separate macros is overhead. Such overhead is incurred by the inability to store results **in registers** between statements. This is not an unavoidable disadvantage of macro-implemented languages, **but** in the initial development for the Oznaki Project no attempt was made to overcome it. Apart from this, the overall structure of the resultant program code is highly optimized. Where applicable, macro statements merely expand to subroutine calls to library software, automatically appended to individual programs if required (e.g., I/O routines, table lookup and computed GoTos). The coding of the tests and branches involved in the structure IF ... **THEN** .. . ELSE . . . ENDIF is in most cases equivalent to **hand-**produced code. (The system implemented fails to utilize compaction possibilities that arise when the same variable occurs in different **sub-expressions.)** At the source program structure level the **procedure-**calling and generating macros permit multiple **entry** points to routines, and allow optimal use **of byte-**saving construction employed widely in hand-assembled code.

The HELP system

The ideas presented above have been put together as a comprehensive design strategy, called **"HELP** for microprocessor software development...[12] **HELP** involves using an advanced mini (or mainframe) as the host machine in a multi-target development system. It involves the code partition scheme described previously, with the algorithmic code implemented in a macro-based higher-level language.

HELP involves a technique for program testing which we call high-level emulation. The traditional emulator creates an image of the target-machine memory space and registers. It then takes each instruction of the target-machine code and executes it interpretively. Thus in the conventional emulator there is a one-to-one relationship between states of the target machine and of the host representation. With high-level emulation, the initial development, testing, and debugging of the algorithmic code is carried out on the host machine, using an appropriate Library of macros. Whenever such a program is run, there are two possible categories of bugs: •

- low-level bugs—errors in macro definition; and
- high-level bugs—errors in the high-level program.

In both categories syntax errors (largely typos) are possible. It is difficult in a macro-based system to check statement syntax in any global manner and provide the user with specific error messages. However, when syntax errors occur, it is often apparent from the listing,. and invariably the tight binding of source and object code leads to rapid location. When a program written in terms of processor-independent macros is tested and validated, there are two dif ferent products:

- tested code for a particular processor; and
- tested high-level software.

Clearly the way to use high-level emulation is to test the software first on the most convenient processor—which might well be the host machine. If the relevant macros for a given target processor have been previously tested, then the macro-processor can immediately assemble valid code for the new processor. Otherwise, if new macros have to be written for a new processor, as has been the predominant experience in the Oznaki Project, one can expect en assembly to have to debug the new macro definitions. The pattern of software development involving use of high-level emulation is shown in Figures 1 and 2.

The picture of software development just given describes the actual development of much of the Oznaki Project software, which was used in school ties in 1977.[133]' Despite the fact that we carried the double burden of implementing both the development system and the actual application programs, there is no doubt that there were significant savings by proceeding in this way. Henceforth for us, and for anyone who might use the Oznaki development system, there is available an extensive library of debugged macros for the DEC-10, and for the 8080 and 6800 processors. III

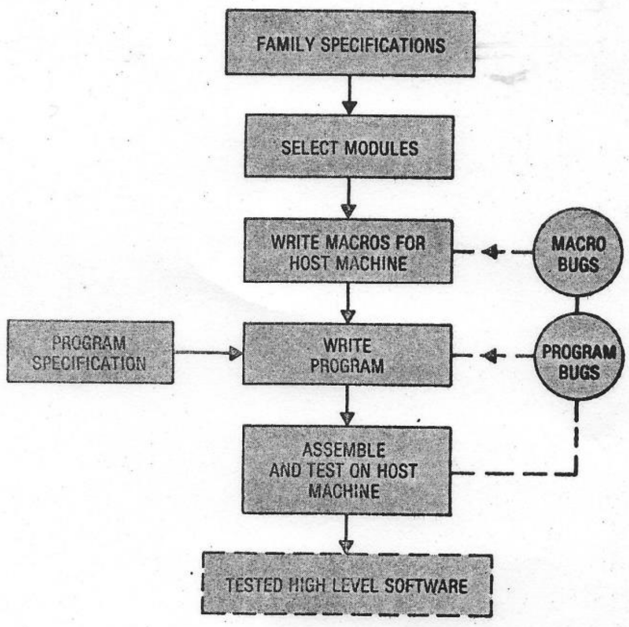


Figure 1. The diagram illustrates the production of tested high-level software for one of a "family" of similar programs. For such a family, a common set of macros can be defined.

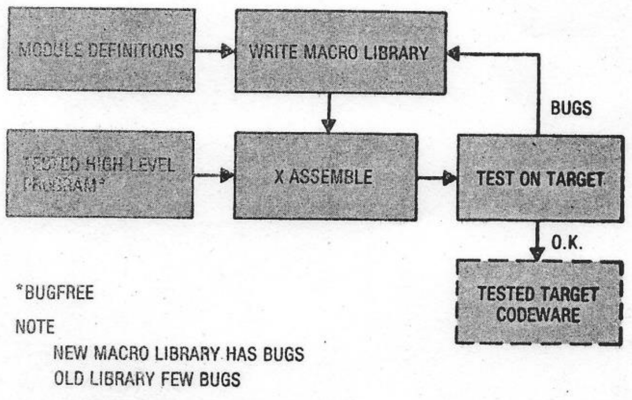


Figure 2. Here, both the input of macro module definitions and of the tested high-level software are the starting points for producing software for a particular target microcomputer. That this works is a result of careful design of individual macro modules. This ensures that any given high-level statement, when expanded for any of the processors, will produce code performing similar functions.

References

1. H. A. Cohen and R. S. Francis, "HELP for Microprocessor Software Development," *Digest of Papers*, COMPCON Fall 77, Sept. 1977, pp. 196-200.
2. *Ibid.*
3. M. H. Halpern, "Towards a General Processor for Programming Languages," *CALM*, Vol. 11, No. 1, Jan. 1968, pp. 15-25.

4. D. E. Ferguson, "Evolution of the Meta-Assembly Program" *CA CM*, Vol. 9, No. 3, Mar. 1966, pp. 190-196.
5. P. L. Evans, "Fast Cross Assembler Production," *Proc. DECUS Europe Conf.*, 1977, pp. 193-195.
6. G. K. Jenkins and P. J. Tyers, "Support Software for Microprocessor Development," *IREECON Int'l Conf. Digest*. Melbourne, Aug. 1977, pp. 122-124.
7. P. J. Tyers, Personal communication.
8. G. S. Herman-Giddens, R. B. Warren, R. C. Barr, and M. S. Spach, "BIOMAC: Block Structured Programming Using PDP-11 Assembler Language," *Software-Practice and Experience,* Vol. E, No. 4, Oct.-Dec. 19'15, pp. 359-374.
9. D. F. Furgerson and A. J. Gibbons, "A High-Level Microprocessor Programming Language," *Digest of Papers,* COMPCON Fall 77, Sept. 1977, pp. 185-188.
10. P. Rechenberg, "MUMS—A Machine Independent Programming Language Consisting of 360 Assembler Macro Calls," *SIGPLAN Notices,* Vol. 12, No. 9, Sept. 1977, pp. 52-59,
11, A. S. Tanenbaum, "A General-Purpose Macro Processor as a Poor Man's Compiler-Compiler," *IEEE Trans. Software Engineering,* Vol. SE-2, No. 2, June 1976, pp. 121-125.
12. Cohen and Francis, "HELP for Microprocessor Software Development."
13. H. A. Cohen, "Oznaki: A New Medium for Mathematicians," in *Learning* and *Applying Mathematics*, D. Williams, ed., Australian Association of Mathematics Teachers, Melbourne, 1978, pp. 274-283.
14. H. A. Cohen and D. G. Green, "Evaluation of the Cognitive Goals of Oznaki: Enhancement of Spatial Projective Abilities," in *ACM Topics in Instructional Computing* (a special publication of ACM Sigcue), A. M. Wildberger and R. G. Montanelli, eds., ACM, New York, 1978, **pp.** 69-90.
15. Cohen, H. A.. "The Oznaki Robotics Language Os," *Proc. 7th Australian Computer Conf.,* Vol. 1, 1976, pp. 128-143.

Bibliography

1. Adamson, W. J., "Macro Processors and Language Extension," *Proc. 7th Australian Computer Conf.,* Vol. 1, 1976, pp. 1-11.
2. Barth, C. W., "STRCMACS—An Extensive Set of Macros to Aid in Structured Programming in 360/370 Assembly Language," *SIGPLAN Notices,* **Vol.** 11, No. 8, Aug. 1976, pp. 31-35.
3. Bennett, R. K., "BUILD: A Primitive Approach to the Design of Computer Languages and Their Translators," *SIGPLAN Notices,* Vol. 11, No. 7, July 1976, pp. 34-40.
4. Brown, P. *J., Macro Processors and Portable Software,* Wiley, London, 1974.
5. Cohen, H. A., "Microprocessor Software Develop. ment Using Macroprocessors," *Proc. 8th Australian Computer Conf.,* 1978.
6. Cohen, H. **A.,** and Francis, R. S., "Programming Constructs for Microprocessors and Bit-Slice Processors," *IREECON Int'l Conf. Digest,* Melbourne, Aug. 1977, pp. 119-121.
7. Cohen, H. A., and Francis, R. S., *Oznaki **Macro Cross** Assembler for 8080, 6800, SC/MP, 6502: Users **Manual,*** Revision 1.5, La Trobe University, **Melbourne,** Australia, Nov. 1977.
8. Cole, A. *J Macroprocessors,* Cambridge **University** Press, 1976.
9. Conrad, M., King, D., and Price, R., "The Integration of Microcomputer Hardware and Software Development Tools and Techniques," *Digest of Papers,* COMPCON Fall 77, Sept. 1977, pp. 201-208.
10: Digital Equipment Corp., *Macro-10 Assembler Programmer's Reference Manual,* 8th ed., Digital Equipment Corp., Maynard, Mass., 1974.
11. Gannon, J. D., and Horning, **J. J**., "The impact of Language Design on the Production of Reliable Software," *Proc. Int'l Conf Reliable Software,* Apr. 1975, pp. 10.22,
12. Intermetrics, A *High Level Programming Language for the Motorola Microcomputer,* (product description—PL/M 6800 product support), Intermetrics, In**c.**. Cambridge, Mass., Dec. 1977.
13. Kildall, G. A., "High-Level Language ***Simplifies*** Microcomputer Programming," ***Electronics,* Vol. 47,** No. 13, June 27, 1974, pp. 103-109.
14. Motorola, *M6800 Microprocessor **Application** Manual,* 1st ed., Motorola Semiconductor **Products,** Inc., Phoenix, Ariz., 1975, section 2-12.
15. Strachey, C., "A General Purpose Macrogenerator." *Computer Journal,* Vol. 8, Oct. 1965, pp. 225-241.
16. Waite, W. M., *Implementing Software for **Non-**Numeric Applications,* Prentice-Hall, Englewood Cliffs, N. J., 1973.
17. Wright, R. J., "Structured Programming in Assembler," *Proc. DECUS Australia Conf.,* 1976, pp. 1017-1019.

Harvey A. Cohen is a senior lecturer in the departments of computer science and applied mathematics at La Trobe University, Melbourne, Australia. His research interests include tiny languages, microcomputer graphics. design strategies in system development, and the nature of problem solving skills. Since 1975 he has directed *the* Oznaki Project, with the support of the Australian Education Research and Development Committee.

Cohen received his PhD in theoretical physics at the Australian National University in 1965. He is the author of a compendium of enigmatic problems—"dragons"--in elementary mechanics.

Rhys S. Francis is a research student in the department of computer science, La Trobe University, Melbourne, Australia. His research interests include macroprocessors, software portability, microcomputer architecture, language design and implementation. His doctoral thesis is in preparation.

Francis graduated with honors in science from Monash University, Melbourne, Australia in 1976.