

COMPANY
CONFIDENTIAL

Engineering News

RECEIVED
TEKTRONIX, INC.

JUL 18 1979

WILSONVILLE
LIBRARY

September 15, 1977 Vol. 4, No. 10 Staff: Burgess Laughlin (Editor) and Scott Sakamoto (Art Director).

Ext. 5674 Delivery Station 50-462

Inefficiency

ASL

**Using High Level
Languages
For Firmware
Development**

HLL

```
FPAR
INTEGER*16: IDX,KEY,CNT,TABLE
ENDF

VAR
INTEGER
ENDV

### INITIALIZE

LOW=1
HIG=CNT

### SEARCH

WHILE LOW<=HIG DO
  IDX=(LOW+HIG)/2
  CASE TABLE(IDX) OF
    [ < KEY ] LOW=IDX+1
    INCREASE LO
    [ > KEY ] HIG=IDX-1
    DECREASE UP
```

Machine Language Instructions

Using High Level Languages For Firmware Development

Larry E. Lewis and J. Lynn Saunders

This article is a later version of a paper the authors submitted to COMPCON. The paper and accompanying slide show will be presented September 6, 1977 in Washington, D.C.

High level languages (HLL) can be powerful tools in firmware development for microprocessor-based products. HLL's are not the perfect solution for all programming problems, but if you're aware of the languages' strengths and weaknesses you can cut expensive firmware development time.

First we will look at the pros and cons of HLL's, then examine techniques you can use to enhance their natural power. And, last, if a HLL is right for your project, we will lay out the features you should look for.

Benefits of HLL's

Low Development Time

"Statements written per unit time" measures programmer productivity without regard to the level of the statements. (Level is the number of machine instructions produced from a statement where the lowest level is one and the highest level might be twelve).¹ In general, the higher the level of the statement, the greater portion of the task the statement implements, hence fewer statements are required. Since the statement rate is constant, coding time is lower with a HLL.

Mind-sized Tasks

When thinking in a HLL rather than an assembly language, the mind is free of mundane, tedious, and detailed tasks such as keeping track of what each accumulator holds. The programmer can focus on the problem rather than on the idiosyncracies of the machine.

Most of the examples in the article use TESLA code. TESLA was conceived by the Software Engineering group (in the Scientific Computer Center) during the summer of 1975. The language is an ALGOL derivative similar to PASCAL. Some of the features of PASCAL have been enhanced and made more friendly. A successful implementation was available in the summer of 1976 and was used by the authors to develop one of the firmware packages described in this article.

The authors wish to congratulate the language designers and implementers for their farsightedness in undertaking such a worthwhile project, and for being receptive to comments and suggestions from users. Tektronix has a distinct advantage in having such a useful tool for the rapid development of firmware.

—Larry Lewis
—Lynn Saunders

Friendly Keywords

Modern HLL's use friendly keywords that closely parallel the English words used to describe the problem. Phrases like IF, THEN, ELSE; DO, WHILE; REPEAT, UNTIL; and CASE, OF are obvious to the programmer. The task of translating thought into a language that the computer understands is easier with a HLL.

Self Documenting

Friendly keywords also make programs easier to understand because HLL source statements are easier to read than an assembly language source listing. And that means a greatly reduced need for comments and possibly elimination of the flow chart phase of the development process.

One method we have used to evaluate performance is to let the programmer's peers read his code. The HLL code can be understood with few or no comments. Figure 1 is an example of HLL code.

Easy to Debug

Errors are proportional to the lines of code produced.² Since HLL's produce more code from the same number of instructions, there will be fewer errors to debug. In our experience, more than one-half of the errors associated with hand-coded assembly language can be eliminated using the compiler of the HLL. For example, keeping track of what is in each index register is an activity that may produce errors in assembly language coding, but is performed automatically by a HLL compiler.

Encourages Good Programming Techniques

Block structured HLL's with rich control structures allow and encourage the designer to use block-structured programming techniques in solving his problems. Such modular design produces more reliable code for a variety of reasons.^{3,5}

Similarity to Programming Design Languages

Some HLL's available for micro-processor firmware development are similar to programming design languages (PDL's) reported in the literature.⁴ Some programmers at Tektronix are successfully implementing the flow chart phase of the firmware design process using PDL's. The PDL's being used are essentially the same, in keywords and structure, as the HLL used in the coding phase. The translation is a trivial one. The programmer feels he has a machine which can execute his flowcharts.

Using a high level language to develop firmware means the product designer can check (through execution) his design sooner. There are a lot of products on the market now that are micro-processor-based, but they have unproven user interfaces. It is very important to check out these interfaces early in the design phase. HLL's make this possible.

Supports Structured Programming Techniques

Many HLL's available for micro-processor firmware development were designed to be easily used with structured programs. Structured programming techniques are much discussed in the literature, and are increasingly accepted in practice.⁵ It has been our experience that using structured programming techniques and a HLL reduces development costs and improves program reliability.

1		PROCEDURE / PROGRAM.ENTRY/
2	V	VAR
3	V	BINARY*8:KEYCODE
4	V	LOGICAL*8:STEPMODE,QEMPTY
5	V	ENDV
6	C	CON
7	C	BINARY*8:STEPKEY/13H/
8	C	ENDC
9	E	EXT
10	E	PROCESS.COMMAND,DISPLAY.INSTRUCTION
11	E	GET.KEYPUSH,DISGARD.KEYPUSH,QTEST
12	E	ENDE
13	R	REPEAT
14	R	IF NOT STEPMODE THEN
15	RI	PROCESS.COMMAND(STEPMODE)
16	RI	ENDI
17	R	IF STEPMODE THEN
18	RI	DISPLAY.INSTRUCTION
19	RI	GET.KEYPUSH(KEYCODE)
20	R	IF KEYCODE=STEPKEY THEN
21	RII	DISGARD.KEYPUSH
22	RII	ELSE
23	RII	STEPMODE=FALSE
24	RII	ENDI
25	RI	ENDI
26	R	QTEST(QEMPTY)
27	R	UNTIL QEMPTY OR STEPMODE
28	R	ENDR
29		RETURN
30		END.

Figure 1. Even without the use of comments, high level language code is much easier to read than assembly language code.

Portability

HLL's are more machine independent than assembly languages. Hence, more code written with a HLL will be portable than it would be using assembly language. Utility routines such as stack and queue manipulators, floating point math packages, and number conversion packages, are common to many projects, so portability is desirable.

Easy to Maintain and Modify

The idea that "once the ROM's are burned, the maintenance is over," is false. Figure 2 shows a ROM package that has undergone seven releases in two years of production. One must not assume that once the product is in production the software task is complete. Life cycle costs for firmware products are similar to those for software products. Our conclusion is that maintenance is as important in firmware as it is in software, and that maintenance is simplified when the programming medium is a HLL.

Objections

This section examines the arguments made against using HLL's for firmware development. Many studies have reached the conclusion that the use of a HLL is warranted only when the number of systems to be reproduced is relatively small.⁶

Small Examples

The examples used by many evaluators of HLLs are too simple to accurately measure how well the HLL will do when applied to a task where modularity and structure are forced on the programmer. The structure of a firmware module longer than 1000 statements is impossible to fully comprehend as a whole. Yet, to make global optimizations, such understanding is critical. Thus the assembly language programmer reaches a point of diminishing returns optimizing his code.

Assumption of User Ignorance

Most benchmarks assume the programmer uses the full power of the language without knowledge of the type of code it produces. In reality, much more efficient code can be produced by intelligent use of the HLL (by keeping in mind the things it does well and the things it does poorly). Our experience shows that two weeks spent learning these kinds of things improves efficiency by an average of 30%.

Failure to Consider Memory Quantum Size

Since the quantum size of physical ROM is doubling every eighteen months, the size of a load module in which the space efficiency becomes important increases at the

same rate. So, with today's technology, if you have a load module which contains 12 kilobytes of code and you have 8 kilobyte ROM's, you stand to gain little by reducing the size of the load module unless you can do so by at least 33%.

Firmware Development Techniques

The techniques we will discuss have proved successful in the implementation of two 20 kilobyte ROM images to be reproduced in quantities of about 1000 per year.

Learn the Things the Compiler Does Well

The designer needs to learn the type of code the HLL produces in order to use those constructs that produce efficient code, and to avoid those that do not. (With our HLL, this was important in passing parameters to procedures.)

If the compiler produces assembly language as object and can produce a listing with the corresponding assembly embedded in the source code, this task of "learning" the compiler is simplified. (The compiler can be thought of as a sophisticated assembler with an extensive, predefined macro capability.) An example is shown in figure 3. Statements 13 and 14 (line numbers 42 and 51) in the figure produce poorly optimized code since this version of the compiler does not recognize constant subscripts. The figure shows statements 13 and 14 replaced with optimized assembly code.

Code Critical Paths in Assembly Language

The ability to execute the compiler's output and measure its performance lends itself well to selective optimization, especially speed optimization. If a loop contains 20% of the code, but consumes 80% of execution time, it probably would be wise to encode parts of that loop in "hand-rubbed" assembly code. Our experience shows, and it is corroborated in the literature, that programs typically spend more than 90% of their time in less than 10% of their code.

An Example

Figure 4 summarizes attributes of five firmware development projects with which the authors are familiar.

Number of Functions is a measure of the overall complexity of the task implemented and correlates with the ROM image size.

Stored Program indicates whether or not the task supports programs definable by the user. Some kind of memory management is implied.

Original ROM	Version					
	2	3	4	5	6	7
A	NEW		NEW			
B	NEW					
C	NEW					
D	NEW					
E	NEW					
F	NEW		NEW			
G	NEW	NEW				
H	NEW					
J	NEW		NEW			
K	NEW			NEW		NEW
L	NEW					
M	NEW					
N	NEW		NEW			
P	NEW			NEW		
Q	NEW		NEW		NEW	
R	NEW					
Changes	16	1	5	2	1	1

Figure 2. Easy maintenance is a key feature of HLL programs because firmware may have to be changed as often as software.

Editing 1 is an indication of whether or not simple editing (like deleting from and adding to the end of the program) of the user program is allowed.

Editing 2 indicates whether or not insertions and deletions within the program are allowed.

Algebraic Calculations indicates whether or not algebraic expressions are processed.

Language Used indicates the programming medium for the firmware development.

RAM Space indicates the size of RAM, in bytes, needed to support the task, excluding user program store.

ROM Space is the size of the executable machine language code masked into ROM.

Development Time is the total time required to design the software/firmware architecture.

Coding Time is the time required to code the task.

Debug Time is the time required to make the resulting code releasable.

The complexity (Number of Functions) of projects 3, 4, and 5 are similar. However, the development time of the HLL programs was one-fourth to one-fifth that of the assembly programs. The space efficiencies are comparable. It is difficult to compare time efficiencies, because all tasks operate in real time environments and all meet the design constraints imposed at product conception.

HLL Features to Look For

Allow Linkage to Physical Address Space

If you are using a HLL for firmware development, you will need a handle on the physical address space. Certain physical addresses such as interface ports and common areas with assembly language routines need to be easily accessible.

The programmer needs embedded assembly statements to make the 80%-20% optimizations we discussed before. Also, if the designer needs a compiler construct which might produce inefficient code, he can embed a small assembly block to perform the same action. An effective way to do this is to enclose the inefficient code in comment delimiters. That enclosed code becomes the comment describing what the assembly language statements do, as shown in Figure 3.

Assembler Compatible Relocatable Blocks

It is our belief that HLL's can successfully be used in firmware development only in the assembler environment. Ideally, procedures should be able to access global variables across procedure boundaries, whether or not the procedure was written in assembly language or HLL. If the compiler produces assembly code as object code, many of these criteria are met. You should be able to easily link the resulting modules into one firmware module.

A

```

1      PROCEDURE /TEST2/
2  V      VAR
3  V      ADDRESS*16:COPYFROM,COPYTO
4  V      CHAR*8:DISPLAY(10)
5  V      BINARY*8:COPYLEN
6  V      ENDV
7  C      CON
8  C      CHAR*8:READYMSG(10)/'READY
9  C      ENDC
10 E      EXT
11 E      COPY
12 E      ENDE
13      COPYFROM=^READYMSG(5)
14      COPYTO=^DISPLAY(10)
15      COPYLEN=10
16      COPY(COPYFROM,COPYTO,COPYLEN)
17      RETURN
18      END.

```

Original source program with statements 13 and 14 highlighted.

C

```

13      # COPYFROM=^READYMSG(5) #
14  A  ASM
15  A                      LDX #READYMSG+4
16  A                      STX COPYFROM
17  A                      ENDA
18      # COPYTO= ^DISPLAY(10) #
19  A  ASM
20  A                      LDX #DISPLAY+9
21  A                      STX COPYTO
22  A                      ENDA

```

Part of the new source code with an optimized, embedded assembly language block for statements 13 and 14 in the original source code.

41	*		
42	*****	13	COPYFROM= READYMSG(5)
43	*		
44	0000 C6 05	TEST2	LDA B #5
45	0002 4F		CLR A
46	0003 FB 0013 1		ADD B 1+LI\$.0004
47	0006 B9 0012 1		ADC A LI\$.0004
48	0009 F7 0001 0		STA B 1+COPYFROM
49	000C B7 0000 0		STA A COPYFROM
50	*		
51	*****	14	COPYTO= DISPLAY(10)
52	*		
53	000F C6 0A		LDA B #10
54	0011 4F		CLR A
55	0012 FB 0015 1		ADD B 1+LI\$.0002
56	0015 B9 0014 1		ADC A LI\$.0002
57	0018 F7 0003 0		STA B 1+COPYTO
58	001B B7 0002 0		STA A COPYTO

Part of the object code produced from the source code in A. It is apparent to the programmer that the code for statements 13 and 14 could be much more efficient.

Figure 3. One technique for using HLL's in firmware development is to learn the type of code the compiler produces and, where appropriate, hand optimize to produce more efficient code. This example shows a sequence optimizing a section of source code.

```

38          *****                               13      # COPYFROM~^REA YMSG(5) #
39          *
40          ENTRY TEST2
41          SECTION CODE$
42          *
43          *****                               14      ASM
44          *
45          0000 2 TEST2          EQU      *
46 0000 CE 0004 1          LDX      #READYMSG+4
47 0003 FF 0000 0          STX      COPYFROM
48          *
49          *****                               18      # COPYTO~^DISPL Y(10) #
50          *****                               19      ASM
51          *
52 0006 CE 000D 0          LDX      #DISPLAY+9
53 0009 FF 0002 0          STX      COPYTO
54          *
55          *****                               23      COPYLEN=10
56          *
57 000C C6 0A          LDA B      #10
58 000E F7 000E 0          STA B      COPYLEN
59          *

```

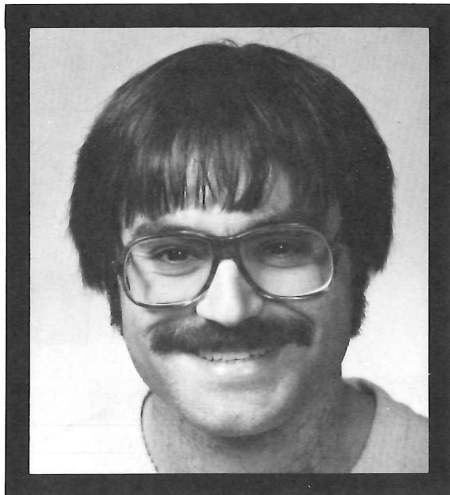
Part of new object code showing optimized code for statements 13 and 14 in the original source program (A).

	JOB 1	JOB 2	JOB 3	JOB 4	JOB 5	JOB 6 (3)
Language Used	ASSY	ASSY	ASSY	ASSY	HL	HL
Number of Functions	10	9	13	45	35	40
Stored Program?	NO	NO	NO	YES	YES	YES
Editing 1 (1)	—	—	—	YES	YES	YES
Editing 2 (2)	—	—	—	YES	NO	NO
Algebraic Calculations?	NO	NO	NO	NO	YES	YES
RAM Space (Bytes)	128	512	1K	1.5K	1K	1.5K
ROM Space (Bytes)	4K	12K	12K	20K	16K	20K
Development Time (Man Months)	10	22	22	42	11	9
Design Time (Man Months)	4	8	7	18	5	4
Coding Time (Man Months)	4	7	8	11	4	3
Debug Time (Man Months)	2	7	7	13	2	1

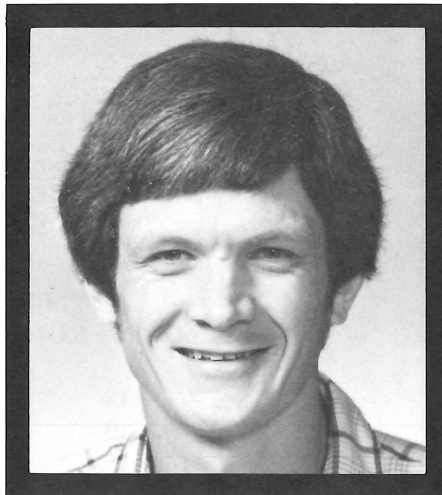
NOTES

- (1) Editing 1 adds these capabilities: display program, add to program, and delete last step.
- (2) Editing 2 adds these capabilities: adds line numbers, arbitrary insert, and delete steps.
- (3) Project 5 is similar to project 4 and uses some of the same design and code time.

Figure 4. Comparing five firmware projects, development time using HLL's was about one-fourth as much time as for assembly language projects. Complexity of the programs and space efficiencies were about the same.



Larry Lewis, software design engineer, TM500 Engineering.



Lynn Saunders, software design engineer, TM500 Engineering.

Timing Readout For 7D01 Logic Analyzer

When using a logic analyzer in the timing mode, it is often useful to know the exact time between events occurring at the probe. A timing readout feature can be added to the DF1-7D01 to automatically measure and display time differences. This feature is especially useful when many time measurements must be made.

HLL Features Cont.

Rich Control Structures

There are several types of control structures appropriate to top-down, structured design. They should be a part of the HLL used in the firmware development process. A minimum might be IF, THEN, ELSEIF, ELSE, and DO, WHILE.

Provide Good Listings

The form of listing produced by the compiler is important. There is a need for assembler type mnemonics that depict the code produced from a particular construct (as in figure 3) immediately after the construct, with clear indications of the associated physical addresses and their contents.

A block nesting indicator for each executable source statement is also useful.

For More Information

If you need more information about using HLLs to develop firmware, call Lynn or Larry on ext. 6640.

FOOTNOTES

- ¹ F.P. Brooks, Jr., *The Mythical Man-Month*. (Reading, Mass.: Addison-Wesley Publishing Company, 1975).
- ² "Update", *Computer*, IEEE Computer Society, Oct. 1974, p. 7.
- ³ Ed Yourdon and Larry L. Constantine, *Structured Design*. (New York: Yourdon, Inc., 1975), pp. 391-392.
- ⁴ Stephen H. Cain et al, "PDL—A tool for software design," *Tutorial on Software Design Techniques*. (IEEE Computer Society, 1975), pp. 172-177.
- ⁵ Victor R. Basili et al, *Structured Programming Tutorial*. (IEEE Computer Society, 1975).
- ⁶ Paul Rosenfeld, "Is there a high-level language in your microcomputer's future?," *EDN*, 20 May 1976, pp. 62-67.

BACKGROUND READING

- Donald J. Reifer, "Automated Aids for Reliable Software", *Proceedings of the 1975 Conference on Reliable Software*, (IEEE 1975), pp. 160, 171.
- F.T. Baker, "Organizing For Structured Programming", *Lecture Notes in Computer Science, Programming Methodology*, (New York: Springer-Verlag, 1975), pp. 38-86.

Other Methods

Other time measurement methods can be used with logic analyzers, but they require some calculating and/or data recording (mentally or on paper). With the LA 501, for example, the user can count the tick marks between events and multiply by the sample interval switch setting. Or, with the 7D01, the cursor can be set at the be-

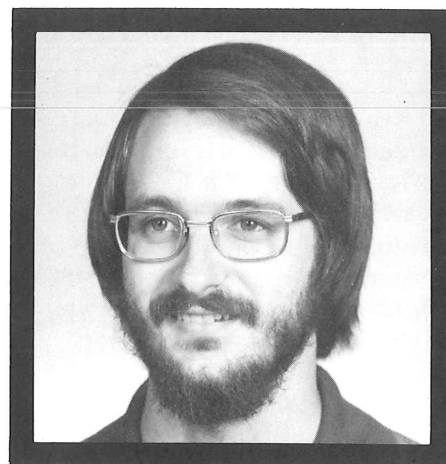
ginning and end of the interval being investigated and the number of samples past the trigger recorded for both cursor settings (T1 and T2). The time difference is then calculated by subtracting the first reading from the second (T2-T1), and multiplying the result by the sample interval switch setting.

With the timing readout feature added to the DF1-7D01, time can be measured between the trigger word and cursor or between two cursor settings. The added hardware reads the position of the 7D01 Sample Interval switch and routes that information to the MC6800 microprocessor in the DF1. The time measurement (in msec, μ sec, or nsec as appropriate) is read out in the upper right corner of the crt.

Example

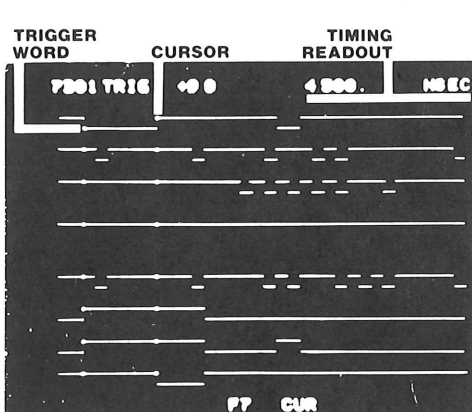
In the top waveform of figure 1, the time between the trigger word and the cursor is measured and displayed on the crt (4500 nsec). What about a pulse that is not related to the trigger word, (one that occurs before or after the trigger word)? First, the cursor is set to the beginning edge of the pulse (figure 2). This first cursor setting is referenced, or set, to zero (figure 3) with a redefined DF1 push-button called "reference time." As the cursor is moved from the zero-reference point, the microprocessor updates the displayed time difference. In figure 4, the difference between the first and second cursor settings is read out as 4.6 μ sec.

Who to Call

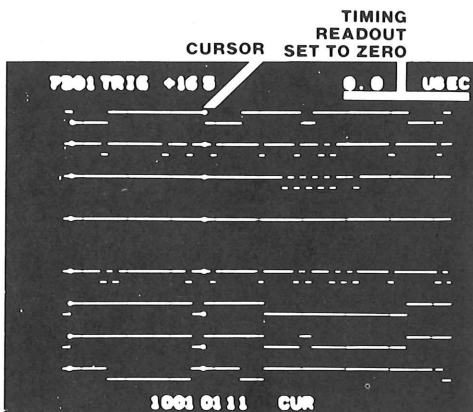


Bruce Ableidinger, project engineer for the DF-2.

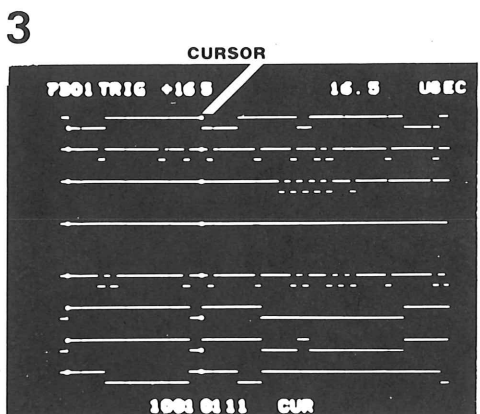
The timing readout mod does not alter any of the instrument's other operating modes. The mod does require some wiring and the addition of one IC to the 7D01 and one EPROM board to the DF1. For more information, contact Bruce Ableidinger at 39-135 or ext. 6995.



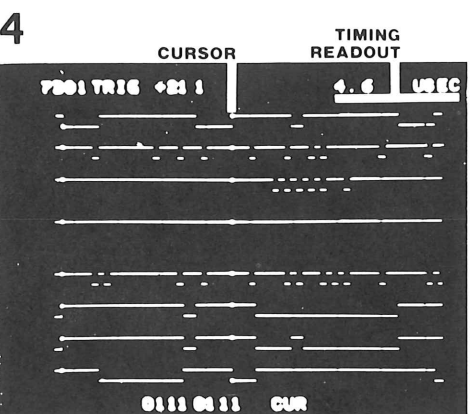
1



2



3



4

Figure 1. Time measurement between (intensified) trigger word and (intensified) cursor. Timing readout is displayed in upper right corner.

Figure 2. First cursor setting for pulse width measurement.

Figure 3. First cursor setting is referenced or set to zero for pulse width measurement.

Figure 4. Cursor is moved to end of pulse (second position). Timing readout displays the time difference between the two cursor settings.

Call for Papers

Components Conference

Five-hundred word abstracts and extended outlines are invited for review for the 28th Electronic Components Conference (April 24-26, 1978 in Anaheim, California). The conference is jointly sponsored by the Components, Hybrids, and Manufacturing Technology Group of the IEEE and the Electronic Industries Association. Abstracts and outlines are due October 28, 1977 and the final papers are due February 13, 1978. Papers are invited in these categories:

discrete components
interconnection and packaging
hybrid microcircuits
reliability, evaluation and failure analysis
materials
manufacturing technology

Intermag Conference

Here's your chance to visit Italy. The IEEE Magnetics Society and the Associazione Elettrotecnica Italiana are sponsoring the International Magnetics Conference which will be held in Florence May 9-12, 1978.

The sponsors are inviting two-page digests (not abstracts) in all areas of applied magnetics, related magnetic phenomena, and information storage technologies. The digests are due December 15, 1977 and the final papers (about six pages long) are due March 15, 1977.

The Technical Information Department (50-462, ext. 5674) is available to help you to contact the conference people, and to prepare your abstract, outline paper, and slide show. We provide editing, typing, illustrating, and coaching (for your talk).

Circuits and Systems Conference

One week before next year's ELECTRO conference, the IEEE Circuits and Systems Society will sponsor the 1978 International Symposium on Circuits and Systems in New York City on May 17-19.

Papers are invited in these broad areas:

- new concepts and novel approaches to the analysis and design of circuits and systems.
- computer-aided techniques for analysis and design.
- new devices and circuits including modeling, analysis, design and applications in signal processing, communications, instrumentation, and control.

There will be two kinds of papers. Regular papers will be up to five pages long and short papers will be one page long. Papers are due October 1, 1977.

A Mailing List for Software People

As a first step in assessing software resources at Tektronix, the Scientific Computer Center is inviting all software people to sign up on the software people's mailing list. The mailing list will provide a communications network for the software community at Tektronix. Those who have signed up on the mailing list will receive the planned software newsletter and announcements (of upcoming seminars, for example) of interest to Tektronix software people.

I want to be on the software people's mailing list.

Name: _____

Delivery Station: _____ Ext.: _____

If you are a software person and would like to sign up on the mailing list, call Roy Carlson on ext. 7668 or fill out and mail the coupon on this page to Roy Carlson at delivery station 50-454.

Feedback

We are always interested in receiving feedback from our readers. Feel free at any time to call (ext. 5674) or drop by (50-462) to let us know what you would like to see in **Engineering News**. We have provided a questionnaire here to make the job easier if you would prefer writing out your comments.

1. Do you keep back issues (in whole or in part)? yes ☐ no ☐
2. What article in the last year has been most interesting to you? (Title or subject). _____
3. Which cover has been the most pleasing to you? (Subject or issue number). _____
4. Would you rather see **Engineering News** go into greater depth than it does now? Or cover more topics? Or both? deeper coverage ☐ wider coverage ☐ both ☐
5. Do you like the **Engineering News** format (readability and overall appearance). Rate us on a scale of 1 (bottom) to 10 (top rating): _____
6. Overall, have we met your need for engineering news? Rate us on a scale of 1 (bottom) to 10 (top rating): _____

7. Which of these regular features are valuable to you?

	very valuable	so-so	little or no value
Calls for papers			
In-prints (notice of Tektronix authors who have been published).			
Metric conversion at Tektronix.			
New technical standards.			
Reprints of papers by Tektronix engineers.			
New technology at Tektronix.			
Special design file (a running catalogue of designs, to avoid reinventing the wheel).			
Announcements of classes and seminars.			
IEEE legal and educational program news.			
New Tektronix products.			
News about GPIB technology.			
Product safety engineering requirements and services.			
Organization charts of engineering and support groups.			
A letters-to-the-editor column for engineering topics (professional or technical).			

8. Additional comments? _____

Please mail the questionnaire to delivery station 50-462. _____

**RETURN MAIL TO
50-462
TECHNICAL INFORMATION**

Maureen Key
If you have
moved, please call Ext. 5407
60 553