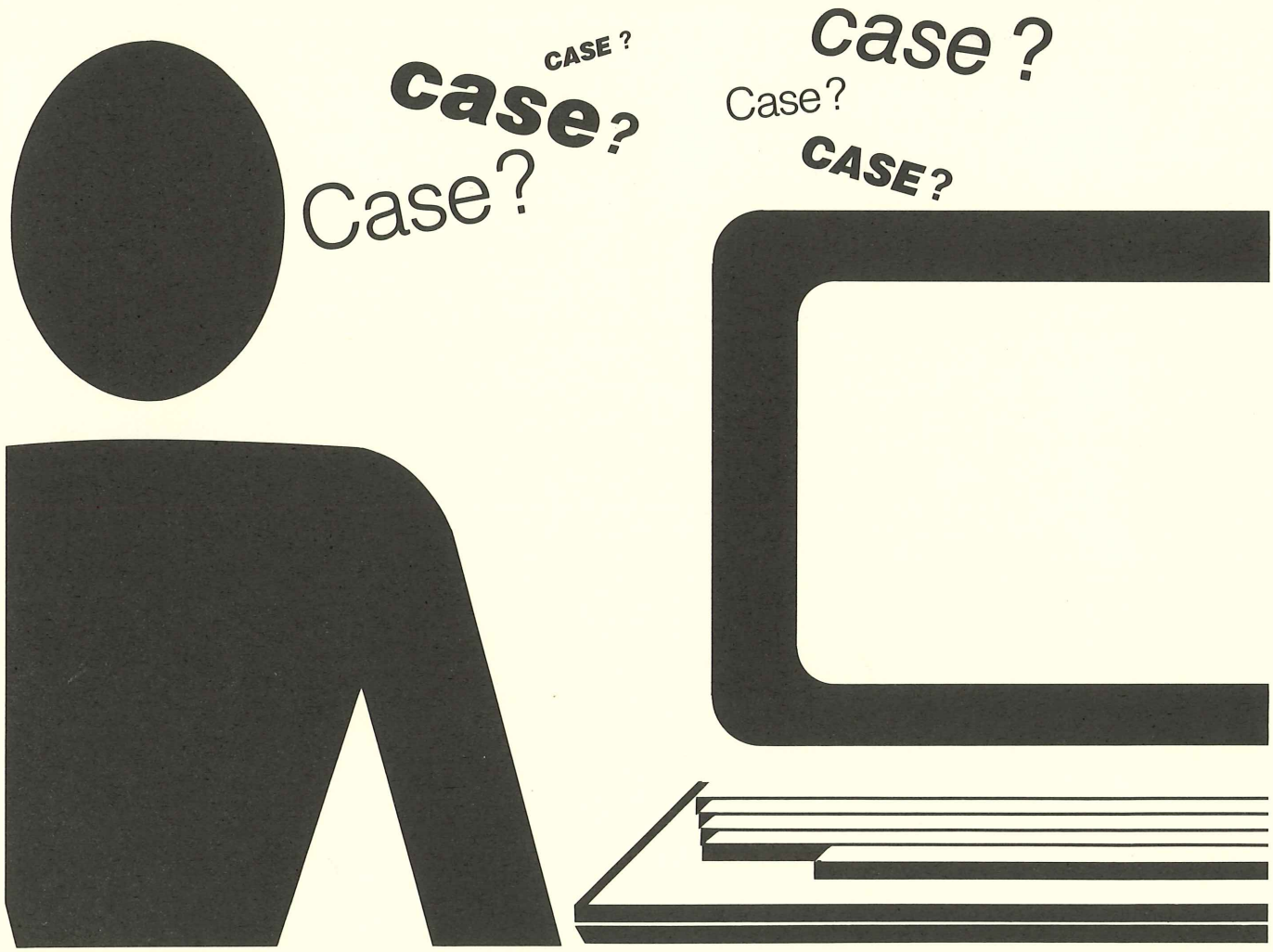
TEK INFORMATION FOR TECHNOLOGISTS

AUG/SEPT 1985

TECHNOLOGY report

USING NATURAL LANGUAGE IN HUMAN/COMPUTER COMMUNICATION



Tektronix®
COMMITTED TO EXCELLENCE

CONTENTS

Volume 7, No. 4, August/September 1985. Managing editor: Art Andersen, ext. MR-8934, d.s. 53-077. Cover: Darla Olmscheid; Graphic illustrator: Nancy Pearen. Composition editor: Sharlet Foster. Published for the benefit of the Tektronix engineering and scientific community.

Why TR?

Technology Report serves two purposes. Long-range, it promotes the flow of technical information among the diverse segments of the Tektronix engineering and scientific community. Short-range, it publicizes current events (new services available and notice of achievements by members of the technical community at Tektronix).

HELP AVAILABLE FOR PAPERS, ARTICLES, AND PRESENTATIONS

If you're preparing a paper for publication or presentation outside Tektronix, the Technology Communications Support (TCS) group of Corporate Marketing Communications can make your job easier. TCS can provide editorial help with outlines, abstracts, and manuscripts; prepare artwork for illustrations; and format material to journal or conference requirements. They can also help you "storyboard" your talk, and then produce professional, attractive slides to go with it. In addition, they interface with Patents and Trademarks to obtain confidentiality reviews and to assure all necessary patent and copyright protection.

For more information, or for whatever assistance you may need, contact Eleanor McElwee, ext. 642-8924. □

WRITING FOR *TECHNOLOGY REPORT*

Technology Report can effectively convey ideas, innovations, services, and background information to the Tektronix technological community.

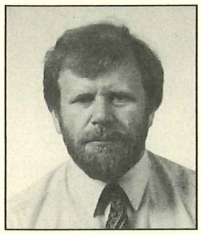
How long does it take to see an article appear in print? That is a function of many things (the completeness of the input, the review cycle, and the timeliness of the content). But the minimum is six weeks for simple announcements and as much as 14 weeks for major technical articles.

The most important step for the contributor is to put the message on paper so we will have something to work with. Don't worry about organization, spelling, and grammar. The editors will take care of that when we put the article into shape for you.

Do you have an article to contribute or an announcement to make? Contact the editor, Art Andersen, 642-8934 (Merlo Road) or write to d.s. 53-077. □

Colorless Green Ideas Sleep Furiously

Natural Language and Human–Computer Communication



Brian Phillips is a principal software engineer in the Computer Research Lab, part of Tek Labs. He joined Tek in 1983 from Texas Instruments. Brian's experiences include a stint as a Research Fellow at the University of Papua–New Guinea, where he taught the first computer science course in that country and learned Tok Pisin ("Pidgin"). He has a BSc in physics and a Post-graduate Diploma in electronic computation from the University of Leeds, England. Brian also holds a PhD in linguistics from the State University of New York at Buffalo.

The Computer Research Lab has been working on a natural-language system for computer systems. Having computers fluent in, say English, would break some of the barriers now discouraging casual users. Even computer experts would benefit. However, natural languages are intricate in their workings, making natural-language projects a "sticky wicket." ...*Just think of how a computer might have trouble with that one.*

As I glance at the screen of my Magnolia workstation, I can see examples of various UNIX commands I have used:

```
lpq -Plw
chips
ls
ms -Tlw encyclopedia &
emacs encyclopedia
```

I know what they mean because I use them frequently and so remember them. There are also many commands I know exist but seldom use. For these I usually can't remember the proper syntax and have to consult on-line documentation or a manual. And, unfortunately, there are certainly commands that I don't know of even though they could be useful to me.

The situation described above is not peculiar to me; it arises whenever a user has to learn a language to communicate with a computer, a language whose uneven use results in uneven retention.

Professional computer scientists can accept the need to learn computer languages to accomplish their tasks. On the other hand, there is the growing class of "casual" users who want to use the machine for some task but don't want to waste time learning a computer language; they see that as a distraction from the task they wish to perform.

One solution for both professional and casual users would be a language they do not have to learn and do not forget. An example lies in what we use routinely with fellow humans; why can't we use our native languages, English in my case, with machines? This argument is attractive. With such capabilities, the user could engage his machine in interactive fact-finding or control, the user could, for example, retrieve information from a database, or set parameters for a simulation without having to learn a formal language designed to suit the task.

Unfortunately it turns out that *Natural Languages*—English, Chinese, etc. (in contrast with *Formal Languages*—UNIX, Fortran, etc.)—are complex and it is not easy to create a system that will interpret them. Nevertheless the advantages make people willing to expend the effort in trying.

So far I have mentioned only the conversational use of a natural language. It is also a durable repository of human knowledge, in letters, books, menos. If computers could "read" books and extract the knowledge, many new vistas would open: chemists and engineers, doctors and lawyers would have better access to their libraries; instructional systems would automatically develop tutorial material from the technical literature.

Giving a command or information to a machine is only part of any interaction. On my workstation screen I also find the statements:

```
PID    TT  STAT  TIME  COMMAND
18933  p0  S     0:02  -csh (csh)
18943  p0  R     0:00  ps-x
8334   p3  I     0:04  -csh (csh)
```

I know what some of that means!

It would be more comforting if computers could generate textural responses meaningful to me. These responses could be error conditions, answers to questions, or reports.

But, natural language usage is not without its problems. In this article I want to present its advantages and disadvantages, to outline the difficulties in building a system, and finally to describe a natural language system we have developed in the Computer Research Lab (CRL). For reasons that will be explained, my discussion will largely be confined to the use of natural language to enter commands and questions.

Advantages and Disadvantages

One obvious advantage of a natural language is that it would not have to be taught to the user. Another is that it can be universal, capable of being used across a wide range of applications. In contrast Fortran is not much use for an interactive graphics editor.

A novice learning a new application and a new interface language may welcome postponing learning the formal language and using natural language so that he can focus on the application. Even for the professional, natural languages are often appropriate: We are all casual users of some systems and forget much between our infrequent usages.

Nevertheless natural language is not the answer to all the problems of human–computer communication. A natural language system interposes another level of software between the user and the machine with a corresponding decrease in speed. Expert users will resent this. For them, the greater efficiency justifies learning the formal language. But in the future, computing systems may become so complex that natural language interfaces will be essential even for the expert user.

There are applications for which a natural language interface would be inane, a spreadsheet for example. It is far better to move a cursor to a box and type in "$200" than to type "Move to column 3, row 4 and enter $200."

The Nature of Language

Humans use languages easily. This has led us into a false impression of the simplicity of language. The lesson was learned 25 years ago when, anticipating a shortage of human translators to cope with Russian technical articles, attempts were made to build automatic translation systems. The attempts failed. Although these systems could not produce high-quality translations, they did show how naive we were about language.

It was thought that effective translation was possible by translating the words of one language into those of the other with some rearranging of the order of words. In German, for example, the verb generally goes at the end of the sentence not between subject and object as in English. The realities of translation turn out to be more complex. Consider

Give me the case.

Is "case" a suitcase, a legal case, or a case of cans? All are possible and all could translate differently in the target language. To resolve the ambiguity one has to understand the sentence, that is, to know its meaning in the context of its use. Unfortunately, representing meaning turns out to be a remarkably hard task.

"Semantics" is that part of linguistics that deals with meaning. It is one of the several levels of organization in language. Different schools of linguistic thought have different distinctions, but all generally include:

Phonology—The sound structure: The "ng" sound is never found at the beginning of an English word.

Morphology—The structure of words. The use of affixes to form plurals and past tenses: pest, pests; like, liked.

Syntax—The structure of sentences: A determiner precedes a noun or an adjective, never a verb. Hence "the on lamp red turn" is not syntactically well-formed.

Discourse—How sentences are put together to form coherent dialogues, paragraphs, etc. A paragraph like the following would be disturbing: "Are the peaches ripe? It will require a new crankshaft."...Each sentence has a different topic.

Semantics—The meaning of utterances: "Colorless green ideas sleep furiously" has no meaning (ignoring poetic license.) It's syntax is good, however.

The early workers in machine translation failed to realize the importance of discourse and semantics in their research. As they were working with written texts, phonology was irrelevant. Since then *Computational Linguistics* has been more diligent in its examination of all of the components of language. However complete solutions are not yet available.

In particular, there are still considerable theoretical weaknesses in the machine analysis of semantics, discourse, and phonology. The "case" example showed the problems that have to be faced in the semantic component in a natural-language system. Discourse requires an understanding of semantics and of topical structure. Phonological analysis would enable spoken input to be accepted. To do this it is necessary to segment an acoustic signal into an unambiguous form akin to the string of characters developed by typing in words. Doing this is particularly difficult because of dialects and differences in individual manners of speaking. We probably all pronounce "coffee" differently but the typed form has no variation, hence the greater ease of handling typed input.

One practical approach to solving the "case" problem is to recognize explicitly that the application domain, say, is "law" and to put only one meaning into the system. This is the technique of creating application-specific sub-languages. All present systems are application-specific; there is no natural language system with humanlike pan-disciplinary fluency.

Present commercial systems are confined either to typed command-entry interactions or to limited aids to machine translation. For the command-entry systems the weaknesses just outlined are less evident and pragmatic solutions are possible. In machine translation, difficult problems can be deferred to the user for handling interactively, or in a post-editing phase.

So far I have been talking generally about limitations in our understanding of language. I would now like to show more specific phenomena that cannot be fully treated because of our lack of knowledge. These phenomena further restrict the facilities found in natural-language systems.

Consider the dialogue:

Q: How many resistors are in the bins?
A: 400,000

Q: (a) Are they full?
(b) When did they arrive?

In (a) the pronoun "they" refers to the bins; in (b) to the resistors. We make the correct determination by understanding the questions in the context of the discourse. To do this for question (a), we would have to know that bins can be full but transistors cannot. However, both bins and transistors can arrive. So how is it that resistors are preferred to bins in (b)? The answer lies in discourse structure: resistors are the topic of the first question and continue to be so in the second as there is no indication of a topic change.

The use of a pronoun instead of repeating the noun is pervasive, it is part of the phenomena of "anaphora." In computational systems anaphora is either disallowed or resolved by, say, taking the nearest preceding "appropriate" noun. This is imperfect and not necessarily easy to implement. Practical systems will thus be restricted in their handling of anaphora.

Language allows us to use incomplete sentences, their completion being possible because of preceding statements:

Q: When did Mike Tomlinson join the company?
A: July 1, 1982.
Q: Harry Barton?

The second question is understood to be of the same form as the first with the new name substituted. This is called "ellipsis." Again, in general, ellipsis is hard to describe fully and can be expensive to implement as it involves retaining analyses of the preceding entries.

Another phenomena is "paraphrasing." The following are a few of the ways of asking the same question of a database comprising a company's sales statistics:

What was the income from the truck division last year?
Give me the sales figures for the truck division from last year.
How much did the truck division sell last year?
What did the truck division make in sales in 1984?

If our sub-language permits all possible paraphrases—and offhand there is no reason for choosing one over another—the language system is going to explode in size. So as a practical necessity, many forms of expression will be excluded.

All limitations forced on the language-interface designer create a problem for the user.

How is he or she to know what parts of his normal language the system covers and hence will accept? It is little use to say that "cleft" and "pseudo-cleft" sentences are not permitted; that is the jargon of the linguist and not meaningful to the average user. Training is a possibility—but the whole idea of having a natural-language interface is to avoid training. Training time, if needed, could just as well be used to teach the formal language.

When coverage is limited, a user can type in a query only to have the system reject it without giving diagnostic help. It is little wonder that such systems frustrate users.

INGLISH

INGLISH, the system I am about to describe, attempts to ease the coverage problem for the user. It is a modest, engineering approach that does not try to incorporate such things as anaphora and ellipsis, yet it is a practical system.

INGLISH—for INterface enGLISH [1]—is written in Smalltalk [2] and runs on the Magnolia and 4404 Artificial Intelligence Workstations. The examples in the figures are taken from a demonstration system that answers questions about the 4404.

INGLISH provides a framework—a parser and display windows—that is tailored to an application by providing a grammar and vocabulary that define the sublanguage of the application.

INGLISH analyzes each word as it is entered and if it does not match any of the expected words of the sublanguage, it rejects the word and pops up a menu of acceptable words.

In figure 1, the user typed "fortran", but this exceeded the coverage and a menu of possibilities appeared. The user selects one and continues to create the sentence. If at any time the user is unaware of how to continue, he or she can type "help" (figure 2) to see the possibilities. Case is irrelevant. However, for clarity we choose to make entries in lowercase and have them echoed in uppercase after INGLISH accepts them.

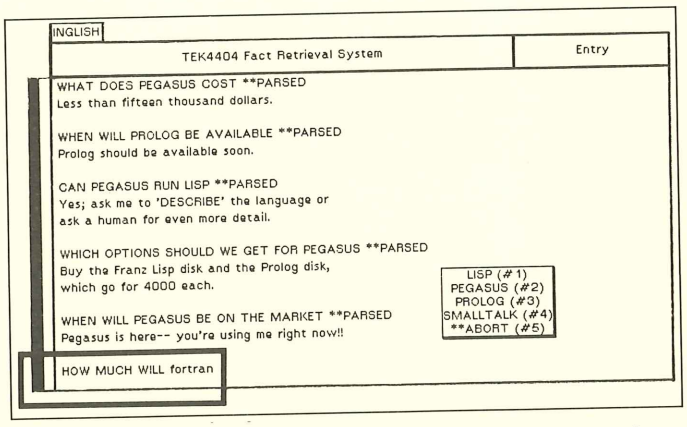


Figure 1. INGLISH, like all natural-language systems, has limited *coverage*—that is, most words and word meanings are not covered. Here the user has exceeded coverage by typing in "Fortran". The system provides a menu of covered words for the user to use in completing his or her query.

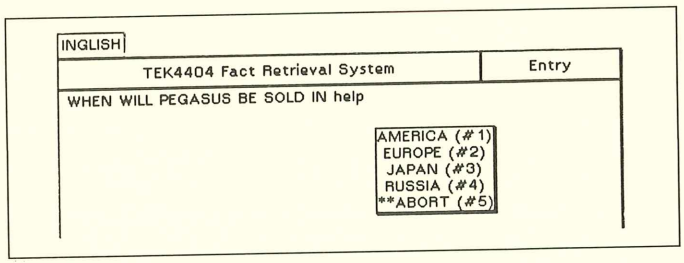


Figure 2. Invoking on-line help to complete a query brings up a menu.

With this approach, the user cannot create a sentence that the system will not accept. A user may stumble through sentence creation, but need never abort the sentence.

There are actually two ways that a sentence can be created using INGLISH: by menu selection or by typing. Human factors experiments[3] have shown that novices prefer the menus, but as experience is gained, typing is preferred. In fact, in INGLISH, the two forms can be mixed within a single sentence; so self-paced migration away from menu use is encouraged as the user becomes conditioned to system coverage. The assistance is invoked only when the user deviates from the sublanguage; otherwise it never intrudes.

In figure 3, the menu type of interaction is shown. A mouse click brings up a menu, figure 3A; a word is selected, figure 3B; and the selected word is added to the sentence, figure 3C. This cycle is repeated until the sentence is completed, figure 3D.

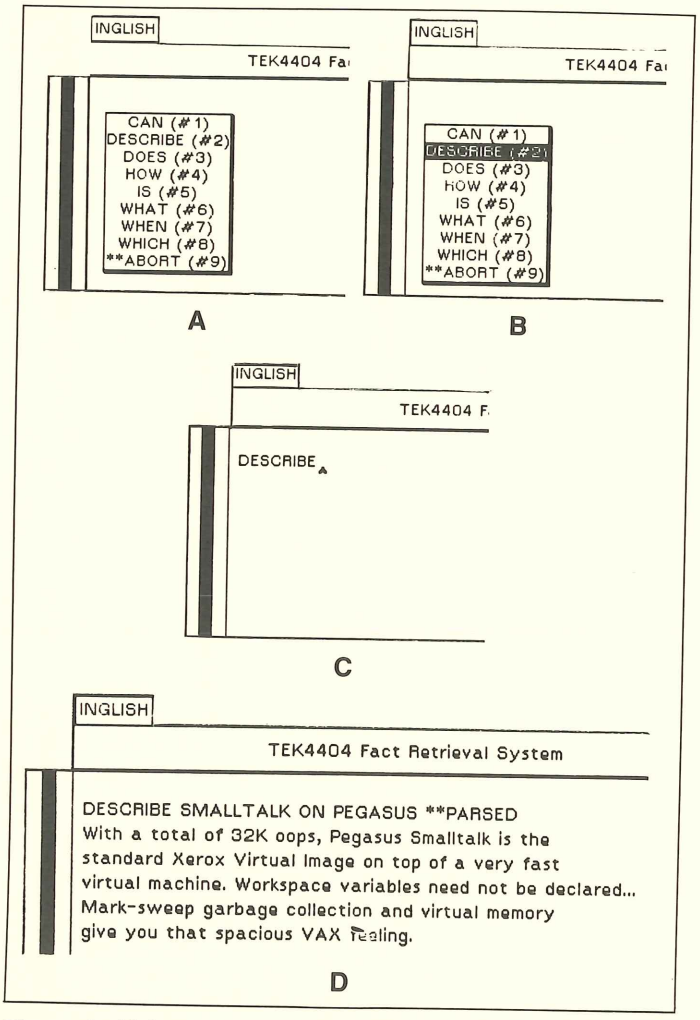


Figure 3. Using INGLISH the user can form a query by typing in words or by using a mouse-summoned menu as has been done in A. In B, "DESCRIBE" has been selected with the mouse as the the first word of the user-created sentence (query). The selected word is copied into the developing sentence in C and the completed query is answered in D.

INGLISH has several other features to assist the typist-user. It incorporates a spelling corrector. Thus in figure 4, "descibe" is corrected to "DESCRIBE." If the current state of a partially entered sentence can be continued in only one way, INGLISH will automatically add this continuation. However, a speedy typist might also be typing identical words; INGLISH will "soak up" such duplication; in figure 5, INGLISH added "WILL" and "will" was also typed but the latter "will" will be deleted. Word completion is also possible. Just enter sufficient initial letters to identify uniquely a word, and a "?" to get the full word. If the identification is not unique, a menu of the matches will appear (figure 6).

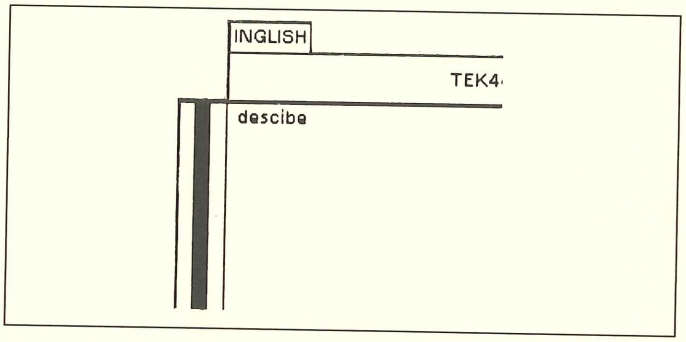


Figure 4. The user can form a query by typing in words too. Here, the user has erroneously typed "descibe". No problem, INGLISH will substitute "DESCRIBE" (not shown).

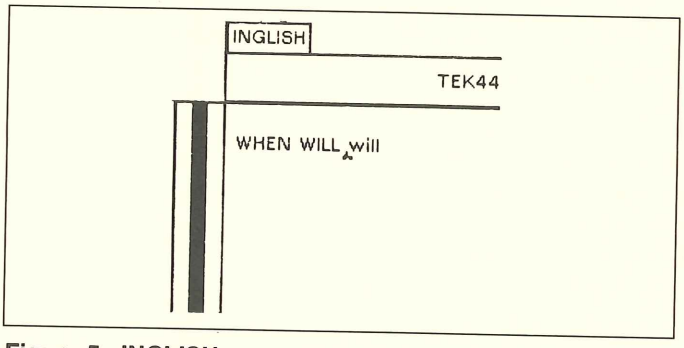


Figure 5. INGLISH can anticipate a word. Here the user has typed "WHEN" and the system adds "WILL". If, during this automatic phrase completion, the user had also typed "will" only one "WILL" will be used—INGLISH soaks up the duplicate.

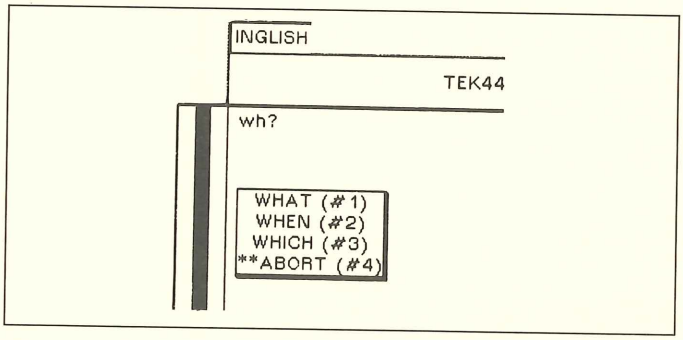


Figure 6. Here, the user is asking for help in completing "wh." A menu of choices responds to the "?"

Everything so far described takes place in the *entry* mode.

As an alternative to attempting to analyze ellipsis, INGLISH offers an *edit* mode. Clicking the mouse in the top right-hand box of the window changes the mode to edit. In the edit mode, the user can use Smalltalk's editor to modify any text in the window to create a new sentence. Suppose the user wanted to ask "When will Smalltalk be available?" Rather than entering the whole query, an earlier entry may be chosen for editing, if this entails less effort. This is illustrated in figure 7. The user noticed that the second question was "When will Prolog be available" and chose to edit this, replacing "Prolog" by "Smalltalk." the new sentence is then selected using the mouse; the shaded box around the sentence indicates the selection. Choosing the menu command "DOIT" causes the selected sentence to be analyzed. As the analysis proceeds, the sentence will be copied to the bottom of the window (after "...which go for 4000 each") and will be followed by the answer. Editing requires only slightly more effort of the user than if the system offered ellipsis, but this type of editing structure makes INGLISH, itself, a much simpler program.

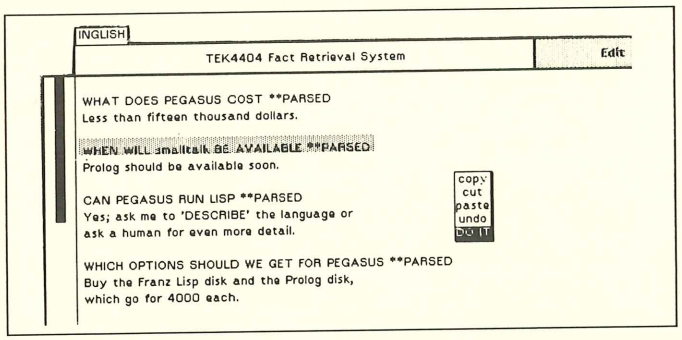


Figure 7. In the edit mode shown here, the user is going to ask about smalltalk availability. The user has typed in "smalltalk" where he or she had used "prolog" earlier. The prolog availability here will be replaced with smalltalk's availability when DO IT is executed.

The parser in INGLISH uses a modified left-corner parsing algorithm[4]. A sublanguage is defined by a *semantic grammar*[5]. This grammar formalism combines syntax and semantics by using categories in the grammar that have meaning connotations. This enables both syntactic and semantic "well-formedness" to be checked simultaneously. (Figure 8 shows a portion of the grammar used in the above examples.)

```
<Sentence> → DESCRIBE <DefiniteItem>
<Sentence> → <WhDefiniteTime> WILL <DefiniteItem> BE <BeModal>
<MachinePhrase> → <Machine>
<DiscountPhrase> → A DISCOUNT <Quant>
<SupportPhrase> → INCLUDE <TimeAdv> <RepeatedDiscreteServiceActivity>
<WhilePhrase> → WHILE RUNNING <Language>
<CanClause> → <Person> GET <Information>
<HowComplement3> → <MachinePhrase> RUN <IndefiniteFeature>
<IndefiniteFeature> → <Language>
<DefiniteFeature> → <Language>
<ExistentialFeature> → <DiscountPhrase>
<Information> → <Language>
<Information> → <VacuousSelector> INFORMATION
<HowSelector> → <TimeSelector>
<HowSelector> → <PriceSelector>
<DefiniteItem> → <Language> ON <MachinePhrase>
<Quant> → FOR <MachinePhrase>
```

Figure 8. A portion of the *semantic grammar* used in the previous examples.

A more linguistic system than the one we used in INGLISH would have separate forms of representation for syntax and semantics. The grammar would describe sentences in terms of nouns, verbs, and the like. The semantics would give a general description of relationships between concepts. Using a semantic grammar is a restriction, but there are still a wide range of applications within the power of INGLISH.

INGLISH in an Application

We are using INGLISH within the Computer Research Laboratory (CRL) to develop INKA (INGLISH Knowledge Acquisition[6], a system to acquire knowledge for a troubleshooting expert system (see *Technology Report,* Sept/Oct 1984). Through INKA an experienced technician can build the knowledge base for the diagnostic process.

INKA constrains the user to enter statements in the sublanguage GLIB (Generalized Language for Instrument Behavior[7]). GLIB was developed within the troubleshooting-assistant project. If the expert was not constrained but allowed to use unrestricted English, automatic translation into the internal form of knowledge of the knowledge base would be impossible.

We Will Offer INGLISH as a General Package

In the light of our experience with INKA we are rewriting INGLISH to make it available as a general package. We will provide a format for describing the sublanguage of an application and for translating INGLISH statements into the internal form required by the application.

For more information, call Brian Phillips 627-1119 (50-662). □

References

[1] Phillips, B., and S. Nicholl, "INGLISH: A Natural Language Interface," TR CR-85-30, Tektronix, Inc., Beaverton, Oregon, 1985.

[2] Goldberg, A., and Robson, D., *Smalltalk 80: The Language and its Implementation,* Reading, MA, Addison-Wesley, 1983.

[3] Gilfoil, D.M., Warming up to Computers: A Study of Cognitive and Affective Interaction over Time, *Proceedings of the Human Factors in Computer Systems Conference,* Gaithersburg, MD, 1982, pp. 245-250.

[4] Griffiths, T., and Petrick, S.R., "On the relative efficiency of context-free grammar recognizers," *Comm. ACM,* 1965, 8, pp. 289-300.

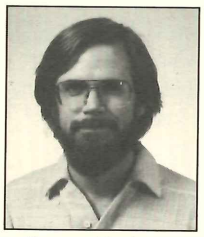
[5] Burton, R.R., "Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems," Tech. Rep. 3453, Bolt, Beranek, and Newman, Cambridge, Massachusetts, 1976.

[6] Phillips, B., Messick, S., Freiling, M., and Alexander, J., "INKA: The INglish Knowledge Acquisition Interface for Electronic Instrument Troubleshooting," TR CR-85-04, Tektronix, Inc., Beaverton, Oregon, 1985.

[7] Freiling, M., J. Alexander, D. Feucht, and D. Stubbs, "GLIB—A Language for Representing the Behavior of Electronic Devices," TR CR-84-12, Tektronix, Inc., Beaverton, Oregon, 1984.

An Expert System Helps Troubleshoot the Wave Solder Process

Sabah Randhawa is an assistant professor of industrial engineering at Oregon State University. Production control, decision-support systems, operations research, and artificial intelligence are his areas of interest. Sabah holds a BS in chemical engineering from the University of Engineering and Technology, Pakistan, an MS in industrial engineering from Oregon State University, and a PhD from Arizona State University. He is a member of IIE, ORSA, CASA/SME, and Alpha Mi Pu.



Bill Barton is a systems/software engineer in the Laboratory Instruments Division. He joined Tek in 1978. As part of the test engineering department, he heads a project that provides information-management tools to the division and investigates how artificial intelligence systems can be applied to manufacturing processes. His previous work includes designing a PC-based quality-information system and developing diagnostic software. He has worked in computer configuration, interfacing, and maintenance. Bill is a member of AAAI and ASQC.

Productivity substantially depends on machine operators' knowledge. How an operator acquires that knowledge directly or indirectly involves experts. Unfortunately this acquisition of knowledge is often inefficient or delayed. Expert systems may resolve these problems. By being friendly, nearby sources of know how, expert systems have demonstrated much promise in helping operators become self-sufficient resolvers of commonplace machine problems.

In any manufacturing operation, machines are idle too often, up to 90 percent of actual production time in extreme cases. As much as 50 percent of this idle time may be downtime. Even at lesser levels of idleness, the loss of production due to down time is a serious and universal problem.

The downtime situation is difficult to rectify largely because the demand for expert maintenance technicians exceeds the supply. Many believe expert systems can relieve this shortage and, ultimately, increase manufacturing productivity.

Our broad objective was to design and develop a prototype of an expert system that would help operators troubleshoot the wave soldering process. Wave soldering is a process critical to producing circuit boards.

Because Tek makes a wide variety of circuit boards, conditions for wave soldering differ. Because boards are made in small batches, soldering process conditions are changed often, as many as several times a day. Even when things go smoothly, each change of process eats up production time. When problems surface, substantial machine and operator time is lost. This time can be significantly reduced.

The first objective in the project was to develop a tool to assist operators solve problems quickly, by themselves, without calling for managers and experts to help. No waiting. Fix it now, and get on with the job is the concept. If this were possible, not only would production efficiency rise, the pool of maintenance experts could be put to assignments more challenging than fixing the same old problems.

A second objective was to develop a tool for training new operators for the wave solder machine. Because operator turnover is high, new operator training is a recurring and substantial load on managers and experienced operators. A powerful training tool could free these resources for more direct productivity.

The System

The two objectives were realized in an expert system that would both train a new operator and later help that operator troubleshoot problems in the wave soldering process.

We developed the expert system iteratively, following these steps:

1. As a test bed, this project had to represent a typical manufacturing process. With the help of Mike Freiling, an artificial intelligence researcher from CRL, we selected a wave solder machine having the problems typical for most machines. The machine is in Building 47.

2. We researched most of the knowledge to be coded in the expert system from manuals and documents and from human experts. What we got from interviewing the experts was heuristic knowledge, that is the subjective rules that characterize the decision making of experts. Our experts were Bert Adams, a manufacturing engineer, Ed Langerveld, who managed the wave solder process, and the experienced machine operators.

3. We identified the causes of most machine malfunctions and product-quality problems:

   - Insufficient solder
   - Voids
   - Solder bridges
   - Icicles
   - Peeling solder resist
   - Nonwetting
   - Warping
   - Rosin film
   - Removed print
   - Raised components
   - Solder balls
   - Missed joints
   - Front edge of boards poorly soldered
   - Rough solder
   - Flooding

   The knowledge we obtained at this step is, for the most part, generic to most wave solder machines. Definitions for the above are found in reference.[1] An example definition is as follows:

INSUFFICIENT SOLDER: holes not full, poor wicking, or several leads not soldered, affecting the integrity of the solder joint.

The process variables, which cause those problems include:

- Solder conveyor speed
- Wash conveyor speed
- Flux temperature
- Flux concentration
- Solder bath temperature
- Solder contamination
- Bubble fluxer air pressure
- Airknife angle
- Wash solution temperature
- Wash solution concentration

We formulated the information gained as inference rules. Such an inference rule states what can be inferred about machine operations from the symptons exhibited by the process variables. Figure 1 shows how information is organized when formulating an inference rule.

4. The software model based on the information identified in step 3 was developed on personal computers—IBM XT and COMPAQ PLUS using PROLOG. (CRL provided help with program refinements.) The features of the expert system include:

Simple, interactive program—The wave-solder expert program is simple and interactive so that operators with little or no computer background can use it.

Operator–system dialog—In a computer-initiated dialog, the user responds to questions asked by the computer. To minimize input errors, user response is kept short. Where appropriate, a menu of valid answers to a question is provided for user response.

It explains its "reasoning"—The system can explain the reasoning process behind the queries and answers in the model.

Questions ordered by problem frequency—Though the model does not explicitly handle uncertainties such as two solutions with equal probability, we based the order in which the questions are asked on how frequently each process variable had caused a specific problem. These frequencies were derived from our interviews with the experts.

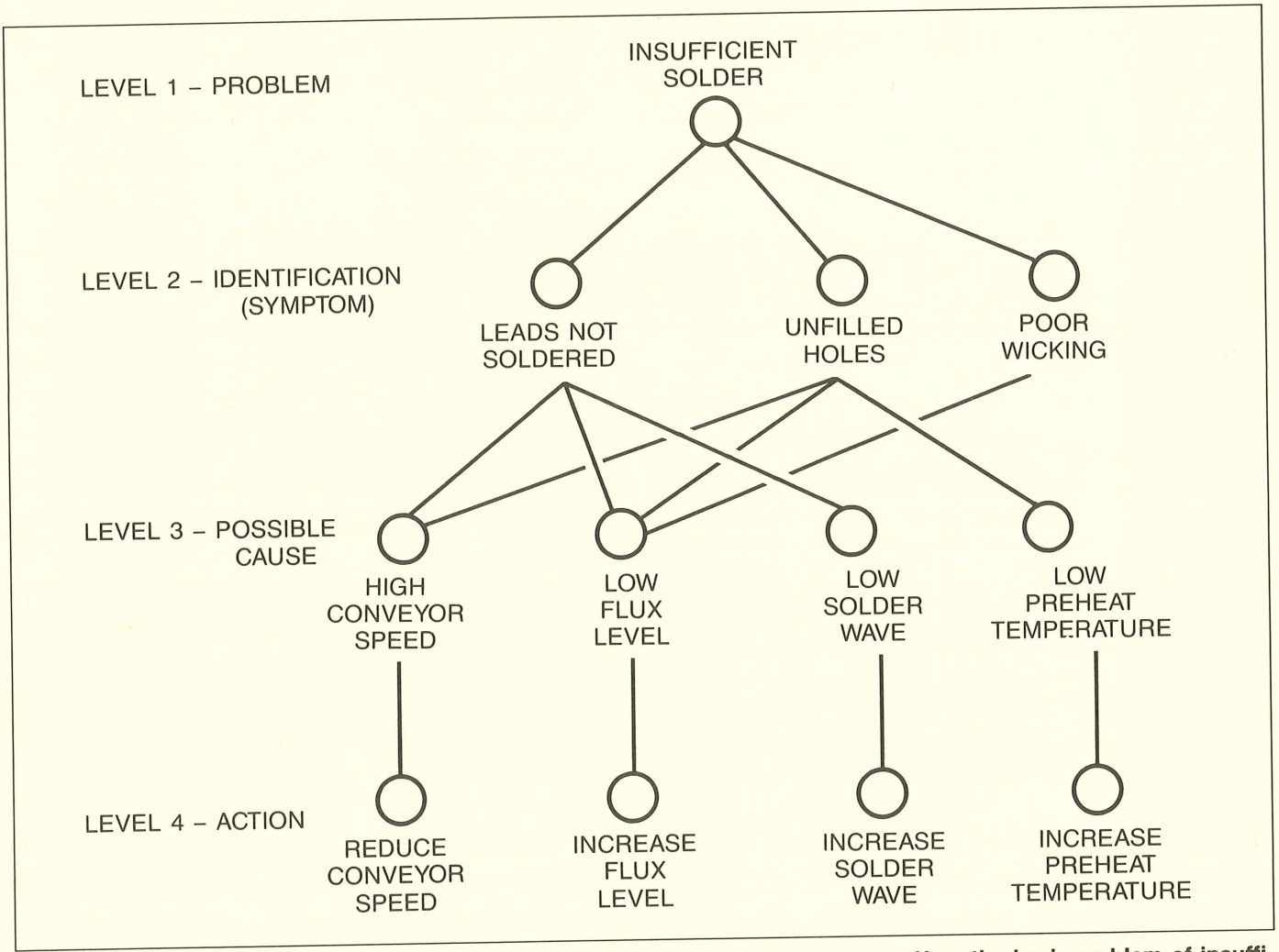


Figure 1. Developing inference rules is a major part of expert-system development. Here the basic problem of insufficient solder is structured by identifying a symptom, one or more possible causes indicated by each symptom, and a recommended action. The operator does not see this structure.

```
                    WAVESOLDER ASSISTANT MENU

        a  Insufficient solder          i  Removed print
        b  Voids                        j  Raised components
        c  Solder bridges               k  Solder balls
        d  Icicles                      l  Missed joints
        e  Peeling solder resist        m  Front edge of boards poorly soldered
        f  Nonwetting                   n  Rough solder
        g  Warping                      o  Flooding
        h  Rosin film                   q  Quit

              SELECT YOUR PROBLEM (a,b,c,...)
a

              SHOW PROBLEM DEFINITION? (y/n)
y

INSUFFICIENT SOLDER; holes not full, poor wicking, or several leads not soldered, affecting
     the integrity of the solder joint.

Return to menu? (y/n)
n

              PLEASE CHECK FOR THE FOLLOWING CONDITIONS:

Several leads are not soldered. (y/n/?)
y

The conveyor speed is too high. (y/n/?)
n

The flux level is less than 1/2 inch above the stone. (y/n/?)
y

Increase the flux level to 1/2 inch above the stone.
Check for plugged stone or air line, if necessary.

Explain? (y/n)
y

Increase the flux level to 1/2 inch above the stone.
Check for plugged stone or air line, if necessary.
     When the following conditions are true:
Several leads are not soldered.
The flux level is less than 1/2 inch above the stone.

              ENTER ANY CHARACTER TO CONTINUE.
:
```

Figure 2. The dialog between system and operator features menus, straightforward questions asked by the system and simple operator responses. "Explain?" and "?" are the expert system's offer to explain its reasoning if the operator is curious or has doubts.

5. The computer system was critiqued by the experts and machine operators for accuracy and usability. They felt comfortable with the system and expressed enthusiasm concerning their participation in the next phase of full implementation.

A complete project description is given in [1]. Figure 2 shows a typical user-terminal session.

Results

The results of the wave solder expert system project are threefold:

First, a test bed was developed to implement long-term evaluation of manufacturing productivity improvements using knowledge-based systems at the operator level.

Second, the system demonstrated that it could relieve managers and experienced operators of some training chores by partially substituting for human tutors.

Third, the experience gained with this prototype helped us develop a general framework for other troubleshooting systems. Lab Instruments will apply this framework to more complex equipment and problems.

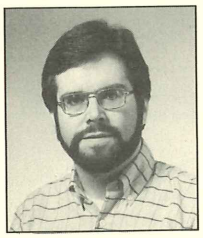
For More Information

For more information, call Bill Barton at 627-4216 (47-589). □

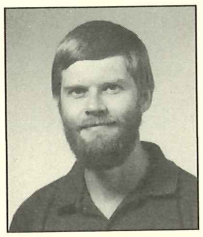
References

1. Randhawa, Sabah, Wavesolder Assistant, Technical Report, Tektronix, Inc., (June 1985).

A Host-Based Prototype Debugging System



Brad Needham is a software engineer in Lab Scopes, part of the Laboratory Instruments Division. Brad joined Tek in 1978 after receiving his BS in computer science from the University of Oregon. At the time this article was written, Brad was a member of the Engineering Computing Systems' Unix-Kernel team.



Alan Jeddeloh is an hardware/software engineer in the Graphic Workstations Division, which includes what was Engineering Computing Systems. Alan joined Tek in 1975 after receiving his BS in computer science from Oregon State University.

How can you debug software without an emulator? An in-circuit emulator is as indispensable to a software engineer as a logic analyzer or oscilloscope is to a hardware engineer, yet there are some projects for which emulation simply isn't available. The development of the 6000 Series workstations was such a project.

To compete in the fast-paced workstation market, we in ECS (now part of Graphic Workstations Division) had chosen to base our product on the most powerful microprocessor available at the time—the National Semiconductor 32000 Series. Unfortunately, the chips were so new that no NS32000 emulators would be available for several years. Rather than build a custom debugging system from scratch, we combined existing software and hardware debuggers into an integrated test and debug system. Although the resulting host-based debugging system did not provide all the features of an emulator, it proved to be a productive environment for firmware development.

Lacking an in-circuit emulator, what methods (however primitive) can be used to debug firmware? One way is to add a serial I/O port and a monitor ROM to your prototype, then connect a terminal to the serial port.

A simple monitor ROM might allow you to examine and load registers or memory by typing register names and memory addresses at the terminal. To transfer control to the monitor ROM from specific points in your test code (an emulator's *breakpoint* and *single-step* functions), you would need to quickly make temporary changes in that code, yet it would be infeasible to burn new ROMs each time you wanted to set a breakpoint! On the other hand, a RAM-based system would have to be reloaded frequently. Loading each word of code by hand is such a time-consuming and error-prone process that it is infeasible for even small amounts of test code.

One solution to the problem of loading RAM is to disconnect the prototype's serial I/O port from the terminal, connecting the line instead to a host computer. A simple debugger could then be written to run on the host machine, accepting commands from a user and translating them into sets of commands to the monitor ROM. You would then be able to tell the debugger to load your test code, and the debugger would send the myriad of monitor commands required to load each word of your code into the prototype.

A host-based debugger offers many productive alternatives. Since the bulk of your debugger is developed on your host machine, the rich software-development environment of your host can be used to quickly create a powerful user interface. Assuming that your prototype's code was generated by host-based tools (compiler, assembler, linker), a host-based debugger can provide sophisticated symbolic-debug by accessing symbolic information stored on the host. All of the features of your host's operating system are available to you: a large, shared file-system; sophisticated terminal I/O control; high-level languages.

Not wanting to write a complete debugger from scratch, we took the approach we had seen used on the MDP 8560 I/O processor project: That is, modify an existing host debugger (such as ADB) so that it communicates with your small monitor ROM. This gave us all the benefits of a host-based debugger plus two more: We had the much smaller task of modifying a debugger rather than re-inventing the wheel. We produced a debugging language familiar to anyone who has used the original host-based debugger—one step toward an integrated development environment.

Features of the Chosen Debugger

The host-based debugger ADB provides sophisticated features for examining and controlling the program being tested:

- Reference to global variables and subroutines by name
- Input of constants in decimal, octal, hexidecimal, or ASCII
- Address and data can be arithmetic expressions involving variables, register contents, and constants (e.g., print the data 20 bytes away from the current top of stack).
- A primitive script language (allowing the writing of scripts to decode data structures)
- Execution breakpoints (via BPT instruction)
- Single-step
- C-language stack backtrace (currently active function calls and their actual parameters)
- Ability to send test data to the program from a host file
- Ability to output data from the program to a host file
- A variety of formats in which to view addresses or data:
  - Two- or four-byte signed or unsigned integers in octal, decimal, or hexidecimal
  - Four- or eight-byte floating point numbers

- ASCII characters or strings
- Instruction disassembly
- An offset from the nearest global symbol

ADB lacks the logic-analysis features of many in-circuit emulators: breakpoints on data references, execution breakpoints for ROM routines, real-time execution trace.

Hardware Support

For host-based debugging, the prototype (figure 1) requires a serial port, the debug-monitor ROM set, and an external character-recognizer card. Serial ports are available on several of the prototypes. On the 6200's I/O processor (IOP), for example, one of its four ports is dedicated as the debug port when in the debug mode and ROM space is sufficient to hold both the debug monitor and the normal bootstrap and diagnostic functions.

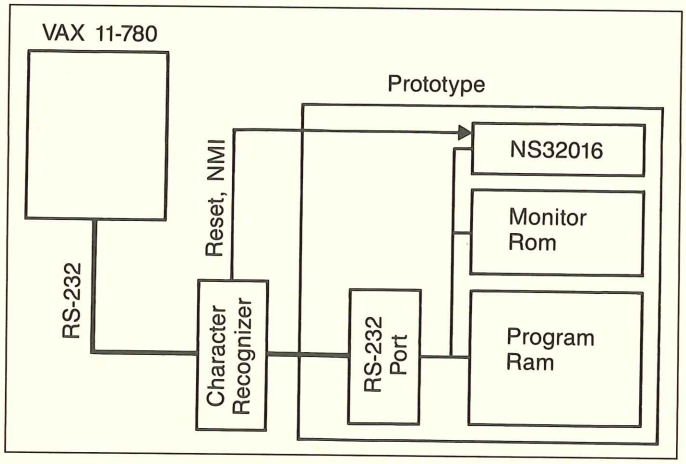


Figure 1. The prototype hardware contains a serial port, monitor ROM, program RAM, and a NS32016 processor. The character recognizer controls the reset and interrupt inputs to the prototype, allowing the VAX-based debugger to load and test protype programs via the RS-232 cable.

Other boards, such as the 6200's compute engines, have neither ROM space nor serial ports. We developed small daughter cards containing a serial port, ROMs, and a small amount of static RAM for these boards. The static RAM on the debug boards allows the monitor to be used even if the main board's dynamic RAM is not functional. Table I summarizes some of the ways in which the serial ports, ROMs, and RAMs were implemented.

Out first prototype had to be interrupted and reset by hand. If the program being debugged lost control, there was no way to automatically return control to the monitor ROM. Taking a few hints from Tek Labs' Magnolia project, we built a simple board that would watch for special characters coming across the RS-232 cable.

The character-recognizer (figure 2) card is an ASCII "filter" that pulses an output line when a given character is seen on the serial line. Up to four different characters can be recognized, each of which can pulse a different output line. The card also has square pins to allow patching RS-232 leads and both DTE and DCE output connectors.

Hardware	Implementation
DB16000	ROM, RAM, and serial port on standard board.
6100 Compute Engine	Monitor shares ROM space with power-up code. Uses system RAM and standard serial port.
6200 "Big Build" Compute Engine	Daughter card with static RAM, ROM and serial port, plugs on using DIN-style connector.
6200 "Production" Compute Engine	Daughter card with static RAM, ROM, and serial port plugs on adapter to CPU daughter card. Four ROMs used to implement 32-bit data path.
6200 I/O Processor (IOP)	Monitor shares ROM space with power-up code. Uses normal sub-system RAM and one of four serial ports already on the board.
6200 I/O Channel (IOC)	ROMs on board. Uses normal sub-system RAM. 61KR02 Sync/Async option board used for serial port.

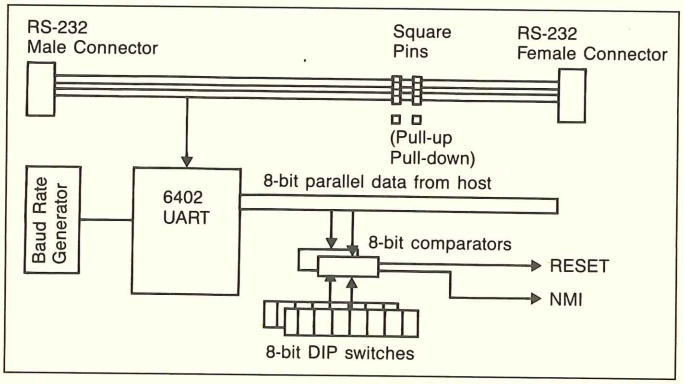


Figure 2. The character recognizer can be set—with 8-bit switches—to recognize up to four ASCII characters. In the debugging process two special characters are used, one to reset and one to interrupt the prototype.

Two of the character-recognizer outputs are used by the debugger. These outputs are patched into the prototype's reset and non-maskable-interrupt (NMI) inputs, usually via square pins on the prototype.

Modifying the Debugger

Choosing the debugger

In many ways, choosing the debugger to be modified was harder than modifying the debugger itself. The two UNIX debuggers available at the start of the project were ADB and SDB. Although SDB offers C-language debugging features—such as printing C structures and breakpointing on C statements—it doesn't provide access to individual machine instructions or registers. ADB, on the other hand, offers good assembly-language debugging but little C-level support.

Because most code for the 6000-series was written in C, we knew that a lot of debugging would involve discovering machine dependencies in that C code. Much of those dependencies would occur at the assembly-language level. Many engineers in ECS were well-acquainted with ADB; few had used SDB. After careful consideration, we decided to base our remote debugger on ADB rather than SDB.

Recently, several programs have been developed that allow both C-language and assembly-language debugging. For example, the 6000 Series' version of SDB includes many of the features of ADB. Third Eye Software, in California, offers a portable debugger called CDB.

Changes for remote debugging

Under UNIX, all debugging of a process is performed by a system call named "ptrace" (for process trace). This single subroutine provides all these features:

- Read/write a word from/to the memory of the process being debugged
- Read/write one of its registers
- Single-step (execute one instruction, then stop)
- Continue running at a given address

All higher-level debugging features use only these primitive operations. Breakpoints, for example, are set by writing a breakpoint instruction into the desired memory locations.

To let the modified debugger access a remote prototype rather than a process on the host, a subroutine named "ptrace" had to be written. This subroutine accepts parameters that tell it to read from or write to a given location in the prototype's memory, to read from or write to a given register in the prototype, or to run or single-step the prototype program. It then passes the request to the prototype's monitor ROM through the RS-232 cable running between the host and the prototype, reads the response from the monitor ROM, and returns the appropriate value to whatever part of the debugger requested the action. (See figure 3.)

The ptrace subroutine provides a complete interface to a prototype once a program is started, but first something has to download the code to be tested. To create a process to be tested under UNIX, a UNIX debugger calls the "exect" system call (for execute and trace); our debugger had to offer a similar function. Again, we wrote a subroutine to replace the function of the corresponding system call—our *exect* subroutine downloads the code to be tested by talking with the monitor ROM.

To interrupt a process being debugged under UNIX, you type ^C. In response to this signal the operating system stops the process and gives control to the debugger. To provide this function in our host-based debugger, we wrote a subroutine that would send a code to the character-recognizer card. Once this card is in place, a prototype can be completely controlled through an RS-232 cable. Invoking the debugger sends the character to reset the prototype (hex81) and typing ^C sends the character to interrupt the prototype (hex82).

Adapting ADB for a foreign machine

ADB was written to debug VAX machine-code. Although an early version of our modified debugger communicated with a monitor ROM, it still believed that the prototype contained VAX instructions rather than NS32000 machine-code. Three parts of the debugger remained to be changed: its idea of the machine register set, its disassembler, and its stack-backtrace function.

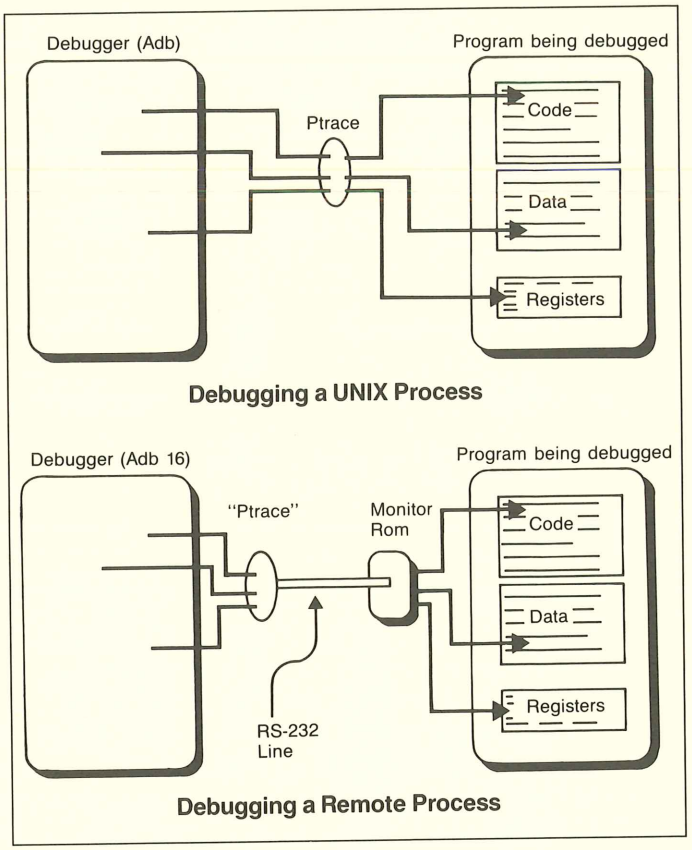


Figure 3. The *ptrace* system call provides the interface between a UNIX debugger and the process it is debugging. In the remote debugging system, a custom "ptrace" subroutine communicates with a monitor ROM to provide the same kind of interface.

Both the NS32000 and VAX processors have a program counter, stack pointer, stack-frame pointer, and several floating-point and general-purpose registers. Changing ADB's knowledge of the machine register set was a straightforward matter. We replaced its table of VAX machine registers with the corresponding table of NS32000 registers.

ADB contains a complete machine-instruction disassembler. That is, for any machine instruction in memory, ADB can print out the assembly-language text of that instruction and its operands. Since NS32000 machine language has almost nothing in common with the VAX, we replaced ADB's VAX disassembler with a NS32000 disassembler.

Two questions commonly arise while debugging C code: "What parameters were passed to this subroutine?" and "What subroutine called the current subroutine?" Since the program stack contains the return addresses and parameters of each active subroutine call, both questions can be answered by examining the stack pointer and contents of the stack. Only a few changes to ADB's stack backtrace function were necessary to accommodate the slight differences in the format of subroutine calls on the VAX and the NS32000.

We avoided one class of problems by choosing the NS32000 rather than, say, a Motorola 68000. On a NS32000, the ordering of bytes within a multibyte integer is identical to the ordering of bytes on a VAX. Both the VAX and NS32000 place the more-significant bytes of a 2- or 4-byte integer into increasing addresses in memory; the Motorola 68000, on the other hand, uses the opposite order. This may seem trivial, but if we had been working with the 68000, this simple difference in byte ordering would have been the source of many bugs.

ADB, like many programs, was not written to be "portable," to be easily moved from one type of processor to another. Every time it collects a set of bytes into an integer value or converts an integer into a set of bytes, ADB assumes that those bytes are ordered as they are on a VAX. Had our prototypes contained 68000s, we would have had to find and fix each of those byte-ordering assumptions in ADB.

Debug Monitor

The debug monitor provides local control for the prototype. The monitor is small (about 8K of ROM) and uses about 1/2K of RAM, including stack space. The monitor is coded in C, except for the module comprising the interrupt/trap handler and context switch, which is coded in assembler. The base addresses used for ROM and RAM vary from prototype to prototype. Load addresses are controlled by using different targets in the monitor's *make* file.

UNIX native compilers and linkers assume that the memory space is contiguous and always writable. The linker assembles the final executable program with the program instruction ("text segment") first, followed by constants and strings ("data segment"), followed by the uninitialized variables ("BSS segment").

When a program, such as the monitor, is to be placed in ROMs, the address space is not always so "normal." The address range for the ROMs may appear anywhere before or after the address range for the RAM.

The compiler group added two flags to the cross-linker to allow specifying the base addresses for the text and BSS segments. We left the data segment to follow the text segment, as it normally does. We set up the checksum/ROM-building utilities to put both the text and data segments in the ROM. This allowed us to use initialized constants and strings in the programs we put in ROM, with the caveat that initialized variables and constants could not be altered at run time.

We designed all prototype hardware (figure 1) so that the debug monitor ROMs get control after power-up or when the host sends the reset character. On prototypes where the ROMs do not normally start at address zero, they are mapped (ghosted) to zero after a reset. The monitor then usually performs some overt act to "de-ghost" the ROMs. For example, on the IOC, the monitor's ghost is disabled automatically as soon as any RAM location is accessed.

The ROM monitors are linked to reside at their normal addresses. The first instruction the monitor executes is a jump. This sets the monitor executing in its normal address range.

The monitor then performs whatever actions are required to de-ghost itself and enable the RAM. The monitor can then also perform other hardware initializations as required by the particular prototype. The specific code is conditionally compiled for each type of prototype, which again is controlled by the targets in the *make* file.

The monitor then performs the rest of the initialization, including setting up the interrupt and trap vectors and setting up a default user-context. The default user-context includes the user-processor registers, a default stack, and interrupt vectors. The monitor then sends its reset message to the host and enters its main command loop.

When the monitor receives control after a trap or an interrupt, it saves the user context, sends an appropriate message to the host, and enters its main command loop. (See figure 4.)

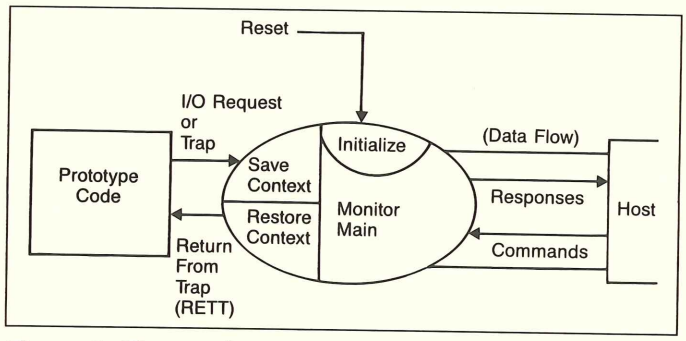


Figure 4. The monitor gets control at power-up or after a system reset. On command from the host, the monitor transfers control to the program in the prototype memory. The monitor regains control when the program makes an I/O request or when a trap occurs.

Once in the main command loop, the monitor sends a prompt to the host and waits for commands. The host may then send commands to read or change registers or memory. Eventually, the host sends a command to either continue execution (go) or single step one instruction (step).

To execute the go command, the monitor first restores all of the user context except the program-status register (PSR), module register (MOD), and program counter (PC). The PSR, MOD, and PC are then loaded onto the monitor's stack, and the monitor executes a *return from trap* (RETT) instruction, which loads all remaining registers from the stack in one atomic (uninterruptable) operation.

The monitor uses the *trace* feature of the NS32000 hardware to implement the single-step command. The NS32000 PSR has two bits that control the *trace* feature: the trace (T) bit and the pending (P) bit.

If the T bit is set at the start of an instruction, the processor sets the P bit. If, at the end of the instruction, both the T and P bits are still set, a trace trap occurs.

When the host sends a *step* command, the monitor sets the T bit in the user's PSR before the user's context is restored. The user program executes one instruction, which then gets a

trace trap, and the monitor then saves the new context and sends a *trace trap* message and a prompt to the host. The major limitation of this method is that the user must be careful not to single step any instructions that load the PSR!

Similar events happen for breakpoints. Before we get to that, let's look at what happens when the instruction being single stepped turns off the T or P bits. The program gets away. Fortunately, our compilers never generate instructions that will turn these bits off. Normally, these instructions will appear in only the sequences initializing the operating systems. Therefore, this potential for turning off the T or P bits turned out to be only a minor inconvenience—provided the user knew of the danger.

Now, look at what events happen for breakpoints.

The host installs a breakpoint by sending commands to the monitor that change a normal instruction into a breakpoint-trap (BPT) instruction. The host first reads and saves the byte at the specified address, then changes the memory byte to a breakpoint instruction. The host keeps a list of which locations have breakpoints set, and what the true memory contents should be. The host installs the breakpoints just before sending the *go* command. When a breakpoint instruction is executed, it causes a breakpoint trap to occur, which returns control to the monitor. The monitor sends a breakpoint-trap message to the host, which then restores the memory locations to their proper contents.

Breakpoints are installed and removed each time the prototype is started and stopped. This makes the prototype memory appear normal when being examined or changed by the user.

Most programs do not need to control interrupt and trap vectors. Test programs usually do not need to (or want to) deal with interrupts and traps. A host-based debug system accepts these programs without major modifications to handle the hardware. Other system software, such as the UNIX kernel, adopt a policy of "peaceful coexistence." System software is set up to determine whether it has been downloaded; if it has, it only commandeers those vectors it requires, leaving the others pointing to the monitor. If the software has started normally, it does the full interrupt initialization.

Debug monitor protocol

The software for the 6000 Series was prototyped using National Semiconductor's DB16000 cards. These were multibus cards, each containing an NS32016 processor and support chips, memory, a serial port, and a multibus interface. The boards carried a debug monitor, which became the model from which the final 6000 Series' debug monitor was built.

These first software prototypes were used to develop and test the compilers, assembler, loader, libraries, and all of the rest of the software-support tools needed. Some DB16000 systems were expanded with extra memory, Ethernet interfaces and primitive disk interfaces and used as the target for the first UNIX port. This allowed software effort to parallel hardware design.

The DB16000 Debug Monitor had several limitations. It was written in assembly language and the source code was not available except as a listing, and the syntax was not compatible with the cross assembler. It also was not particularly portable to other hardware. For these reasons, we decided to write a new monitor for the DB16000s and port it as required to the various prototype configurations. Since by this time we had written a version of ADB that used the National monitor, we retained most of the National command set for compatibility.

The host-monitor communications are similar to normal commands and prompts that might be seen between a host and a user on a terminal: The monitor sends a prompt, the host sends a command, the monitor may send the results of the command, and the monitor then sends a new prompt.

The table below lists the command set provided by the debug monitor:

CHANGE / PRINT MEMORY	
CMB <address> <data>	Change memory byte (8 bits)
PMB <address>	Print memory byte (8 bits)
CMW <address> <data>	Change memory word (16 bits)
PMW <address>	Print memory word (16 bits)
CMD <address> <data>	Change memory double (32 bits)
PMD <address>	Print memory double (32 bits)
F <address> <address> <byte>	Fill memory range with <byte>
CHANGE / PRINT CPU REGISTERS	
CCF <data> / PCF	Change / Print Config register
CFP <data> / PFP	Change / Print Frame Pointer
CIN <data> / PIN	Change / Print Intbase register
CMO <data> / PMO	Change / Print Mod register
CPC <data> / PPC	Change / Print Program Counter
CPS <data> / PPS	Change / Print Processor Status
CRn <data> / PRn	Change / Print general register n
CS0 <data> / PS0	Change / Print Stack Pointer 0
CS1 <data> / PS1	Change / Print Stack Pointer 1
CSB <data> / PSB	Change / Print Static Base
CSP <data> / PSP	Change / Print current Stack Pointer
X	Fast read of CPU context
CHANGE / PRINT FPU REGISTERS	
CFn <data> / PFn	Change / Print general register n
CFS <data> / PFS	Change / Print Status
Y	Fast read of FPU context
CHANGE / PRINT MMU REGISTERS	
CBC <data> / PBC	Change / Print Breakpoint Count register
CBPn <data> / PBPn	Change / Print Breakpoint register n
CEA <data> / PEI	Change / Print Error / Invalidate register
CMS <data> / PMS	Change / Print Status register
CPFn <data> / PPFn	Change / Print Program Flow register n
CPTn <data> / PPTn	Change / Print Page Table Base n
CSC <data> / PSC	Change / Print Sequential Count register
Z	Fast read of MMU context
CONTROL AND HOUSEKEEPING	
Ack	Acknowledgment / synchronization
Go	Start execution
L <checksum> <address> <data> ...	Down-load program
Ok	Data Handshake
Step	Single-step one instruction
U <checksum> <count> <data> ...	User data from Host
Version	Print ROM monitor version

The *Ack* command helps maintain synchronization between the host and the monitor. After the monitor powers up, or anytime after the monitor sends an asynchronous message to the host (such as after a breakpoint trap has occurred), the monitor rejects any further commands from the host until an *Ack* command is received.

The *U* and *L* commands and the responses to the *X*, *Y*, and *Z* commands send their data in a packed binary format. The format packs each three successive 8-bit binary bytes into four ASCII characters. This allows binary data to be transmitted over a normal, 7-bit host connection with reasonable efficiency. Each command is individually checksummed to ensure its integrity.

The *X*, *Y*, and *Z* commands were added to the original command set to speed system response when single-stepping a program. The original debugger polled each register separately each time the debugger regained control. Polling register by register was slow because it required about 35 separate commands and 70 separate reads (one read for the response, one read for the following prompt)!

The *X*, *Y*, and *Z* commands return the CPU, FPU, and MMU registers in just three commands. The register sets are sent to the host as single packed binary blocks. The context is divided into three commands because a given prototype may be lacking an MMU or FPU.

User requests

Our monitor provides four user-request calls to support simple character I/O and simple system control. One of the two drawbacks of the original monitor supplied by National was that it provided no handshake for user writes. This meant that a user program could easily (and frequently) overrun the host computer, causing data loss.

The other drawback of National's monitor was that its user-requests use the *service-request* (SVC) instruction to communicate with the monitor. The SVC instruction causes a trap to a supervisor program; this program is then expected to interpret the user's request, perform the requested action, and return to the user. The difficulty in this was that we were porting operating systems and other code that used the SVC mechanism for its own purposes.

The monitor provides four user requests: *read, write, exit,* and *setconfig.* The user requests are invoked by placing the appropriate arguments in CPU registers and doing a subroutine call to location 10 (hexidecimal). Although this avoids using the SVC call, it does require that the user process turn off interrupts (if they are on) before making the call.

The *read* call causes the monitor to send a read request to the host. The host responds when it has data for the user process by sending the U command containing the data. The monitor unpacks the data into the buffer specified in the user request and returns control to the user process.

The *write* request causes the monitor to send the specified data to the host. After the data is sent, the monitor waits for an *OK* command from the host, indicating that the data has been properly received. This simple handshake prevents the user process from sending data faster than the host can read it.

Both the *read* and *write* requests use the "packed binary" format described above to transmit the data, resulting in a full 8-bit data path from the user process to the host and back.

The *exit* request provides a clean way for the user process to terminate. The request allows the process to pass a termination status back to the host. Unlike other user requests, the *exit* call does not return control to the user-process.

The *setconfig* call provides a way around a shortcoming in the NS32000 design. The processor's *config* register is set by software to specify which optional chips (memory management, floating point, etc.) are present. If an instruction specific to a given chip is executed and the corresponding bit in the config register is not set, an undefined-instruction trap occurs.

So far, so good. But there are two problems: The *config* register cannot be read to determine its current setting, and if a given *config*-register bit is set but the corresponding chip is not present, the CPU hangs in a catatonic state. This means that there is no way for a debug monitor to dynamically determine what the configuration is, and—worse—the monitor cannot tell if the software being run has changed the *config* register to enable a chip that is not present on all prototypes. The *setconfig* command enables the user process to inform the monitor when it changes the *Config* register.

The version of the C-standard I/O library used by the cross loader makes use of the *read* and *write* monitor calls in its I/O subroutines. This made the standard UNIX-input/output routines available to programs running on a prototype. A test program for a math library, for example, could be compiled and run on a VAX, then built using the cross compiler and cross loader, downloaded to a prototype, run, and the results of both runs compared.

Why This Approach Was Successful

In our project, the host-based approach to debugging was successful for several reasons:

(1) We were beginning a new hardware design, instead of modifying an existing one. We could easily add debugging support—a monitor ROM and an RS-232 port—to our boards. Our task would have been almost impossible had the hardware design been frozen earlier.

(2) The software to be tested and debugged had very few real-time dependencies. Our peripherals had very few software timing constraints—data overruns could be easily avoided during testing. The prototype RS-232 driver, for example, could be tested by typing one character at a time on the system console. Had we been developing real-time code, such as firmware for a digital oscilloscope, our task would have been much harder. As it turned out, smart logic analyzers (the 1240 and DAS) were sufficient to make the necessary real-time observations.

(3) The debugging model under UNIX is simple and powerful. It is a straightforward matter to modify any UNIX debugger, replacing the *ptrace* subroutine with a custom subroutine (in our case, one that communicates with a monitor ROM through an RS-232 cable).

Benefits of the Host-Based Debugger

The user interfaces of the remote debugger and the host's debugger were nearly identical—engineers in ECS didn't have to learn a new (and incompatible) debugging language just because their programs were destined to run on a prototype rather than a VAX.

Early prototypes of the hardware were few and had to be shared among many groups. Because remote debugging removed the need for the engineer to be physically near the prototype being tested, scarce prototypes could be pooled. Prototypes could be accessed on an as-needed basis rather than by wandering from lab area to lab area asking "is anybody using your prototype?"

Since the debugger ran on the same host computer used for all of our other tasks, we found it easy to share data: A prototype program could be fed test data from a host file and could store its results in another host file. The test results could be examined or edited by one of the host's many file-processing programs then sent by electronic mail to other members of the project.

Many early tests involved tiny fragments of prototype code. Remote debugging dramatically reduced the time spent to run such short tests; we didn't have to walk to a lab, find an unused prototype and nearby terminal, log into a development system or host computer, and manually reset the prototype. Without leaving his bench, an engineer could swiftly find an unused prototype, download and run a program, and store the results of the test.

Remote debugging allowed engineers to run test programs from almost anywhere, at anytime. It was routine for one of us to call our VAX at Tek in the evening from a terminal at home, compile a test program, load that program into a prototype over at Tek, and then examine test results—all in only a few minutes.

This debugger was just one step toward an integrated program-development environment. In ECS, all of our cross-development tools were based on their corresponding native tools. For example, the only differences between the NS32000 C compiler and the VAX compiler were their names—*c16* versus *cc*—their options and file naming conventions were identical. One coherent set of tools—compiler, loader, assembler, editor, etc.—shared common invocations, flags and switches, file formats, revision control, and build control.

Time to Develop the Debugging System

From the time the project was started until the final versions of the monitor ROM, character recognizer, and modified debugger were available, six months had passed. During that time, we spent approximately five man-months on the project.

For More Information

For more information, call Brad Needham, 627-1583 (39-222) or Alan Jeddeloh, 685-2882 (61-215). □

Bringing IC Vendors to Tek via "CONNECTIONS"

By Wes Bruning, Manager, Connections Program

The Connections Program is a critical part of Tek's strategy for the computer aided engineering market. But Connections is not only a strategy, it's a way for Tek circuit designers to access the products and the know-how of a score of IC vendors.

The Connections program links IC vendors (foundries) with the CAE Systems Division's design tools through *vendors' libraries.* Why are IC-vendor libraries important? Well, not having IC-vendor libraries on a CAE workstation would be like not having applications programs for your computer.

In CAE systems, libraries provide the essentials of circuit design and design implementation: graphics for schematic capture, models for simulation, physical descriptions for layout (place and route), and the details of fabrication. Without the libraries an engineer couldn't capture a schematic, nor simulate, nor fabricate, even with a powerful CAE system.

At this time, we are in the process of developing libraries directly for twelve semicustom-IC vendors, and indirectly for three others. This means each of fifteen IC vendors will maintain Tek-compatible semicustom libraries (graphics and logic models) for the CAE 2000 and its simulators. These libraries are formatted in the *Generic Library* for use with any CAE 2000 simulator. Additional vendor libraries will be added.

The *Generic Library* is a database of simulation models for semicustom ICs.

Each IC vendor, in a joint effort with our applications engineers, creates simulation models for its semicustom family.

AMCC, for example, specializes in ECL gate arrays. A *Generic Library* exists for each of AMCC's four gate-array families—the Q1500, QH1500, Q3500, and Q700. Each library consist of simulation models of each cells in a family. These models reference the primitives (ANDs, ORs, NORs, etc.) specific to a particular CAE-2000 simulator.

Library components or "tool models" are created by substituting simulator-specific primitives into library-specific simulation models. If the designer wishes to simulate his or her library-specific QH1500 circuit with IDEAL, the *Generic Library* simulation model for the QH1500 would compile using the IDEAL tool models. This would produce a "tool deck" acceptable to the IDEAL simulator. If the designer specified the HILO-2 simulator, the Generic Library model (of the QH1500 library) would compile using HILO tool models. Thus a "tool deck" acceptable to the HILO simulator would be produced.

The *Generic Library* simulation models, therefore, have to be developed only once by the library designer (typically the vendor) to enable the IC designer to use multiple simulators. The number of simulators one can use is limited only by the number of simulator-specific primitive sets that exist. At this writing, four simulators are supported: IDEAL, HILO-2, SCALD and Zycad.

That's enough for now about about how libraries and simulators work. Let's talk about Connections as a program and how it helps sell Tek CAE workstations. It does this by providing libraries for the user of a Tek workstation. And Tek designers are—or will be—users!

Who Should Make My IC?

One top concern of the electrical designer or design manager is: Which IC vendor should I choose to fabricate my design so as to achieve my goals for performance, functionality, and cost? Who can give me high-reliability parts and on-time delivery? Clearly, there are many factors involved in selecting a semicustom vendor.

Just as clearly, there are many factors in choosing a CAE workstation. A key factor is the number of libraries in its database. Because the "correct" foundry for one application may be wrong for another, the workstation supported by the most IC vendors has an edge. Not that the workstation with the most libraries always wins, but it sure helps. Connections was implemented to provide Tek's CAE Systems' Division sales engineers and their customers with as many IC-vendor libraries as possible, as rapidly as possible.

The heart of the Connections Program is cooperative selling, team cooperation between the sales engineers of CAE Systems Division and the sales engineers of various IC vendors. These sales teams are already working together, swapping sales leads, making customer calls and presentations jointly. We intend potential customers to see Tek offering both CAE design tools (and associated Tek equipment) and strong ties to IC-vendor manufacturing know-how. In Tek's CAE marketing strategy IC-vendor relationships are key.

Right now, we have such relationships with twelve vendors: AMCC, Gould AMI, ASEA HAFO, Barvon Research, California Devices, Fairchild, LSI LOgic, Mostek, Motorola, NEC, TriQuint, and VTI. We have indirect contacts (through Source III) with three more vendors: IMP, Comdial and NCR. More vendors will be added soon. All such vendors will soon be "on line" to the Tek design engineer designing a semicustom IC. Because Tektronix is a major account to these vendors, they are willing—even anxious—to work with Tek engineers on new designs.

Part of the Connections Program enables IC vendors to present themselves to the Tektronix engineers and managers. We are doing this through seminars and articles. *Technology Report* will be a major vehicle by which technical and vendor information will be passed to the Tektronix designer. Starting with this issue, articles drafted by various IC vendors will be presented addressing technical aspects of semicustom and custom IC design.

For More Information

For more information, call Wes Bruning, 629-1488 (94-520). □

Wes Bruning is the manager of Connections, a program of the CAE Division. Wes joined Tek in 1977 from the Naval Ocean Systems Center where he did hydrodynamics measurements. He is a registered professional engineer whose BSME is from San Diego State University.

It's Time to Look at CMOS/SOS

This *Connections* article is based on material supplied to the CAE Division by ASEA HAFO, a semiconductor company active in research for about 30 years. Doing this article is unusual for *Technology Report*. But *Connections* too is unusual. *Connections* is a CAE Division program that is, among other things, integrating design criteria from many vendors into the CAE database that is at the core of Tek's CAE 2000. Wes Brunning of the CAE Division explains *Connections* elsewhere in this issue.

SOS stands for silicon on sapphire. It's a technology that provides excellent device isolation. As a result, SOS technology excells as the basis for making exceptionally high-quality, high-speed circuits. In combination with CMOS, SOS is a most promising candidate for digital VLSI circuits because these circuits are becoming denser and faster and designers, therefore, are increasing concerned about leakage and stray capacitance.

In an SOS circuit, the isolating monolithic sapphire substrate is used only as a carrier, that is, the structure upon which circuitry sits. To make a SOS circuit, a thin silicon epitaxial film is grown in the sapphire surface and transistors are then fabricated in that film by conventional MOS technology.

The basic feature of SOS is the complete isolation achieved between devices on the same chip. This isolation is achieved by using a sapphire substrate. With CMOS/SOS, designers can produce absolutely latch-up-free CMOS circuits having the highest-possible integration density. It also means very low parasitic capacitances, resulting in higher speed and lower power dissipation than possible with "bulk" CMOS. (Most ICs are made starting with solid-silicon wafers—"bulk" in the jargon.) Another important advantage SOS offers is a high level of design flexibility, employing mixed junction transistors and separate voltages on the same die—these are particularly valuable for custom circuits.

Nothing, including SOS, is perfect, but the relatively minor technical disadvantages of SOS—higher leakage current within a device and higher flicker noise—do not affect digital applications. And the higher cost of SOS materials compared to bulk is not a significant drawback. So, for highly reliable and fast digital applications, SOS is a very good choice.

Why CMOS Isn't More Widely Used

Why then, if CMOS/SOS has these powerful characteristics, hasn't it been more widely adopted by other semiconductor manufacturers? This is the question most frequently asked by companies considering a custom integrated-circuit solution for their product. The evolution of CMOS/SOS provides the answers.

Historically, semiconductor manufacturers have based their business on standard rather than custom components. (They have done custom work mostly as an internal research and development effort.) Standard circuits are produced in volume and have survived well in price-competitive environments.

Until recent years, silicon-on-sapphire circuits have not been price-competitive with bulk. They cost more to make because dicing saws and photomasks, for example, wear out faster. The wafers themselves have been more expensive. And few companies have been competing for the sapphire market. This will change for the better when more semiconductor manufacturers move into SOS to achieve its technical advantages. In manufacturing, projection printing and laser cutters are eliminating the costs of frequent saw and photomask replacement.

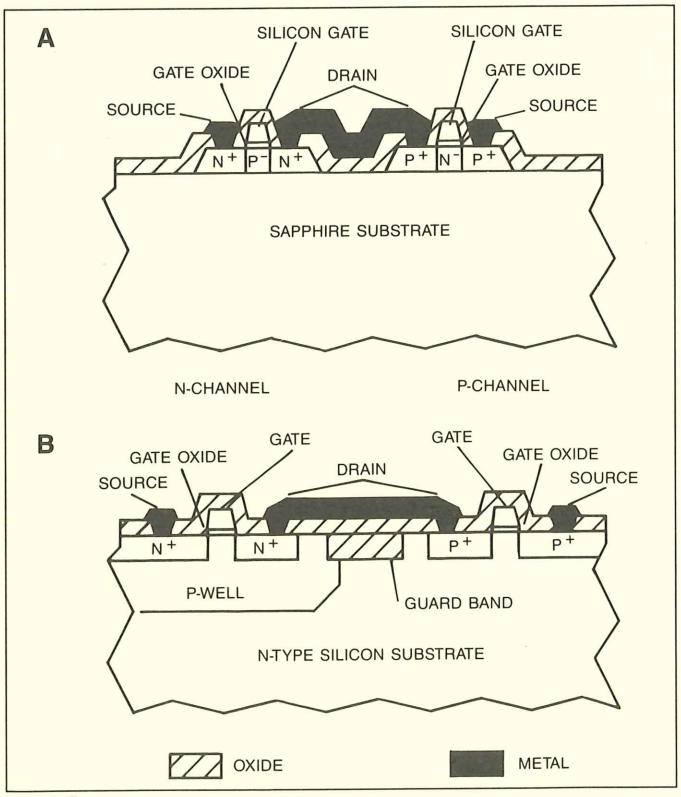


Figure 1. The isolating saphire substrate of CMOS/SOS (A) does not require the silicon-consuming guardbands and the P- and N-wells of bulk CMOS (B).

CMOS/SOS has been a working process for making integrated circuits since 1967, but it is only now that CMOS/SOS has become accepted as a major process. Three factors have forced this acceptance: increasing complexity, the need for fewer design constraints such as latch-up worries and multiple voltages on the same die, and increased demand for application-specific integrated circuits (ASIC).

A few major electronic companies have adapted and improved CMOS/SOS for their own use. They have done so to produce more performance-competitive commercial products or to fulfill stringent military requirements. Most of these self-developed circuits are application specific—that is, ASICs.

Tradition has been a restraining factor too. Most custom houses are break-offs from standard-circuit manufacturers. Their process engineers and designers have carried over their familiarity with the bulk processes to their new employers. Because of this inertia there has been no concerted effort to explore the advantages of SOS for custom applications.

CMOS/SOS came along around 1967, about four years after bulk CMOS, and cannot be expected to achieve market acceptance before or even simultaneously with bulk CMOS. But it is clear that the trend is toward using custom circuits over standard circuits. This will accelerate the use of CMOS/SOS as the best process for custom digital applications (see Table). Using CMOS on a silicon substrate is only the first step; CMOS/SOS is the next.

Comparing Costs— CMOS/SOS Against Silicon "Bulk" CMOS

Let's compare two typical process lines for custom integrated circuit production—one for bulk and the other for CMOS/SOS. To make this comparison fair, we will make these assumptions: Both processes use 3-to-4 micron design rules and 4-inch wafers. And moderate numbers of wafers per design will be run.

The cost of one oxide-isolated silicon-gate bulk CMOS wafer is about $350. A sapphire wafer costs about $450, about 30 percent higher than bulk silicon. SOS has fewer process steps than bulk, but then its blank wafer costs more.

SOS vs. Silicon Bulk CMOS

Parameter	SOS	Bulk
Design Flexibility	Higher	Lower
Speed	Higher	Lower
Power Dissipation	Lower	Higher
Flicker Noise	Higher	Lower
Leakage Current	Higher	Lower
High-temperature Performance	Better	Worse
Latch-up	No	Yes
Radiation Tolerance	Higher	Lower
Packing Density	Higher	Lower
Wafer Material Cost	Higher	Lower
Yield (4-micron)	Equal	Equal
Yield (2-micron)	Higher	Lower

On the other hand, the packing density for the CMOS/SOS circuits is typically 10 percent higher than for bulk. If both processes have identical yields, the real cost difference per chip will favor bulk silicon by about 10 percent; this, however, does not take into account the technical advantages of SOS. Below 3 microns, the situation favors SOS. SOS yields exceed bulk, so at 3 microns and smaller, CMOS/SOS circuits cost less to produce even though sapphire, itself, costs more than silicon.

Comparing Technical Characteristics

Design flexibility and packing density: CMOS is better because of its superb noise immunity, speed, and power efficiency. And CMOS tolerates variations in supply voltage and ambient temperature better than silicon. CMOS/SOS is even better, offering many technical advantages and it is even more design-tolerant than the bulk processes. This tolerance shortens the time necessary to bring an SOS circuit to market. It is the sapphire substrate that makes CMOS/SOS superior. By isolating devices from one another, it eliminates the need—inherent in bulk processes—for guard bands and P and N wells. Both of which consume chip area and restrict layout flexibility.

Because of the "ideal" isolation between devices on the sapphire substrate, diodes can be arbitrarily connected without any risk of forming parasitic bipolar transistors. This is not the case for bulk circuits. It's also quite practical to use several supply voltages for an SOS circuit—again because of the isolation. SOS circuits can be galvanically isolated (pure insulation due to dissimiliar materials) from each other on the same chip, a major SOS feature.

Speed and power dissipation: Basically, these characteristics in a CMOS circuit are set—that is limited—by parasitic capacitances. They are smaller in SOS than in bulk, resulting in higher speed and lower power dissipation in the SOS circuit.

Flicker noise and leakage current: The quality of 600-nm-thick silicon film on a sapphire surface is not as good as that in the other CMOS processes because the match between the silicon and the sapphire crystal is not perfect. This, in turn, creates somewhat more noise and leakage currents in the sapphire circuits, making SOS less adaptive for analog functions. However, digital qualities are not affected except that dynamic logic on SOS needs a higher refresh rate at room temperature than bulk. The difference in refresh rate lessens at higher temperatures.

Latch-up, radiation tolerance and high-temperature performance: Because PNPN junctions always exist in bulk CMOS circuits, applying certain voltages to a circuit could start a thyristor effect (latch-up) that could destroy the chip. No thyristors can exist in SOS. Because of the isolating characteristics of the sapphire substrate, latch-up can't occur, not even when circuit features are scaled down to the fundamental limits. In bulk CMOS, the latch-up problem worsens as circuit dimensions shrink.

The freedom from latch-up means that SOS circuits tolerate transient—momentary—radiation much better than bulk circuits. Such damaging radiation can come from momentary nuclear exposure and solar flares, or from alpha particles in the IC package itself. By changing the process, all CMOS circuits can be hardened to low background radiation. However, for resistance to transient radiation, SOS is superior.

SOS circuits can work at much higher temperatures (300°C) than bulk circuits (125°C) because the complete isolation between the transistors means no leakage current between devices, even at very high temperatures.

Yield: The small mismatch in the interface between silicon and sapphire lowers the yield of SOS to some extent. On the other hand, the SOS yield doesn't suffer from pinhole problems. Pinholes in the oxide under interconnects overlying pure sapphire have no effect on the circuit's functions. In bulk, pinholes are catastrophic. The SOS process contains fewer steps than the silicon-gate bulk process, which, of course, gives SOS a yield advantage. At 4-micron design rules, the yields for oxide-isolated silicon-gate bulk CMOS and CMOS/SOS are about the same per unit area. However, when dimensions are scaled down, SOS yields are better than bulk. As densities increase, this better yield will be a determining factor in the development of wider market acceptance of CMOS/SOS circuits.

For More Information

For more information, call Wes Bruning 629-1488 (94-520).□