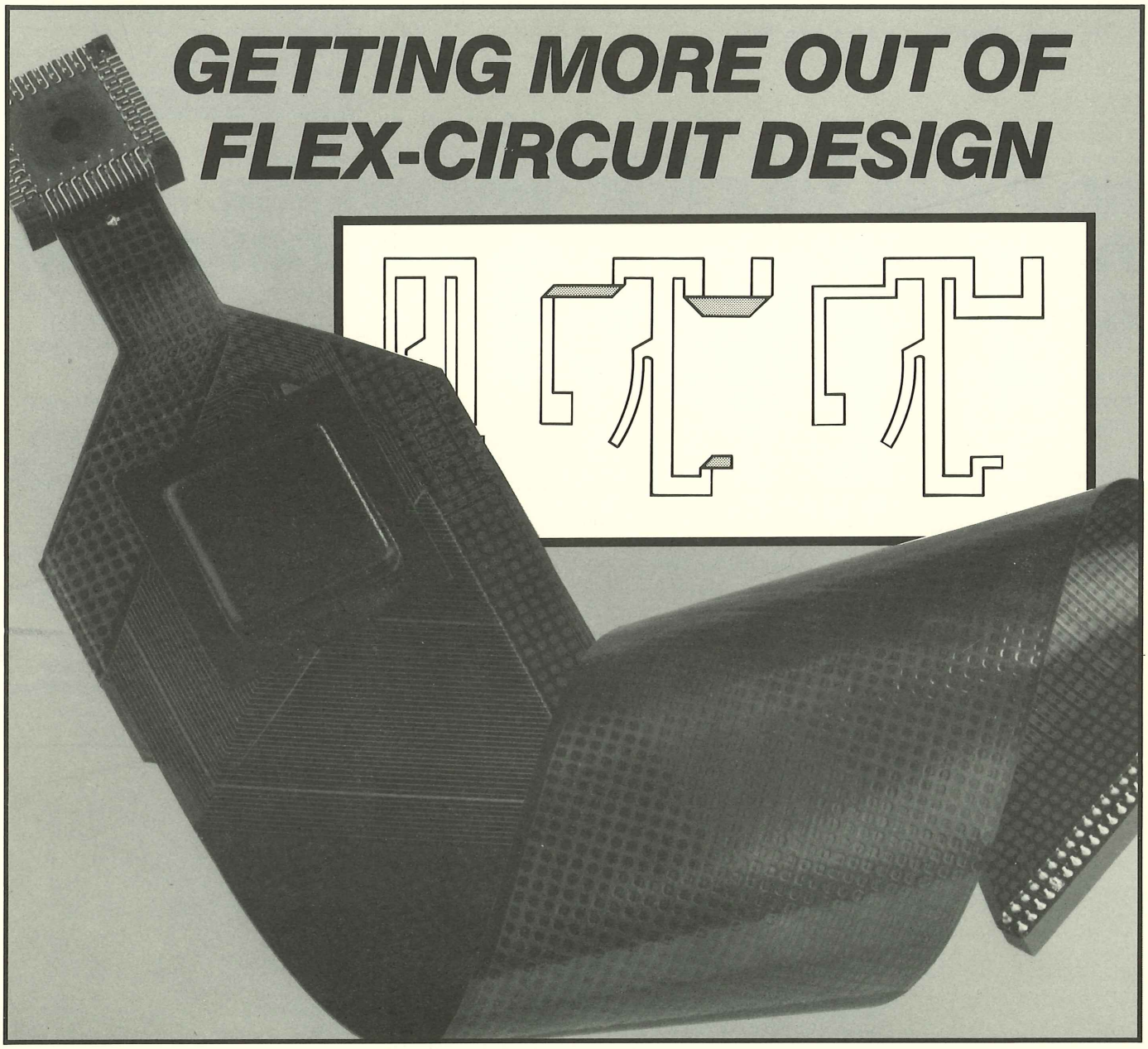
TEK INFORMATION FOR TECHNOLOGISTS

NOV/DEC 1984

TECHNOLOGY report

GETTING MORE OUT OF FLEX-CIRCUIT DESIGN



Tektronix®
COMMITTED TO EXCELLENCE

CONTENTS

Volume 6, No. 8, November/December 1984. Managing editor: Art Andersen, ext. MR-8934, d.s. 53-077. Cover: Nancy Pearen; Graphic illustrator: Darla Olmscheid. 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).

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.

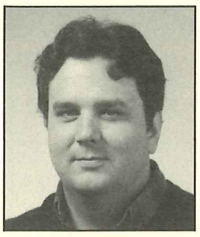
How long does it take to see an article appear in print? That is a function of many things (the completeness of the input, the review cycle, and the timeliness of the content). But the minimum is six weeks for simple announcements and as much as 14 weeks for major technical articles.

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. □

FLEXIBLE CIRCUITS PART I

FLEX CIRCUITS ANSWER A COMPLEX PROBE DESIGN PROBLEM



David G. Payne is an electromechanical engineer in Logic Analyzers, part of DAG. Dave first joined Tek in 1977. He then spent a year at Freightliner as a designer, later rejoining Tek in 1978. Dave has an associate's degree from Portland Community College.

This article illustrates some unique properties of flexible circuitry by showing how a flex circuit was used in a complex 68-conductor probe. With the increased use of more complex integrated-circuit packages, leadless chip carriers, and surface-mounted devices, probe construction and probing techniques are going through major changes. Today's more exotic components require new probe designs. The author, a divisional design engineer, also discusses flexible-circuit design techniques and how they apply to other situations. For the vendor's view, see "Getting More Out of Your Flex-Circuit Design" (in this issue).

Flexible circuits are answering needs in ways never imaginable with wire and cable technology. Generally, flexible circuits are much like printed wiring boards (PWBs), in that their basic function is to route electrical signals, but this is where the similarity ends! Because flex circuits are pliable, that is flexible, they are amazingly versatile (see figure 1). This versatility, plus new shielding materials and extreme circuit densities fit flex circuitry for applications that can't be solved with traditional cables and printed wiring boards.

For example, customers of our division (Logic Analyzers) badly needed a system to probe devices in leadless chip carriers (LCC). The usual probes – built with discrete wires, cables, or clips – would not solve these customers' problems. At best, the old techniques would be too expensive. At worst, they would be cumbersome and difficult to attach.

The Application

We had to provide a way to connect 65 of the 68 pins in a user's microprocessor socket to a remote IC socket of the same type (see figure 2). To be close to the circuit under test, the remote socket had to be on the flex circuit itself. (The remote socket contains the user's microprocessor during testing and must be near the circuit under test to minimize crosstalk.)

Many of the pins on the remote socket had to be connected to the personality module. The three pins in the user's socket not connected to the remote socket had to be terminated at the module; this required routing through the remote socket area, actually bypassing it. The major problem all this posed for us was how to gain connection to the user's socket without compromising the user's probing convenience.

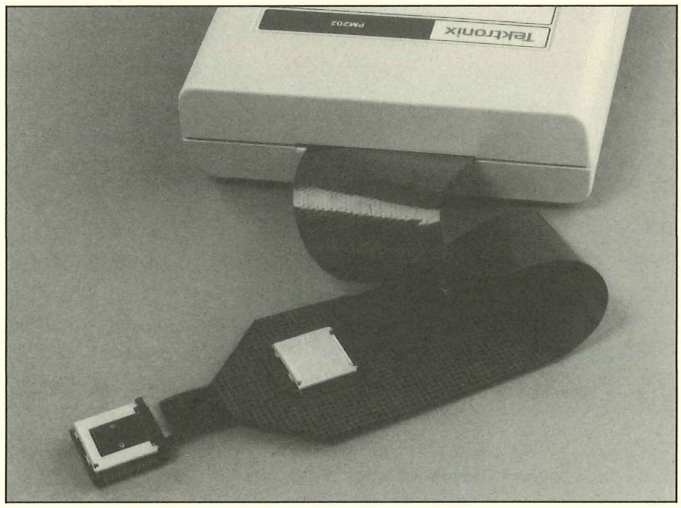


Figure 1. With this flex circuit, 68 conductors can be placed where they are needed to do logic analysis in a microprocessor's actual application environment.

Making the Connection

Our first and most challenging task was gaining connection to the user's microprocessor socket. We decided to fabricate a printed wiring board (PWB) of the same dimensions as the leadless chip, and *somehow* make a high-density, permanent connection to this substrate (the PWB). After much thought about what connection method would be low-profile (a profile of less than .030" was needed), permanent, and reliable – lap soldering seemed to be the only answer.

Lap soldering is a method where exposed (bared) pads on the edge of a flex circuit are folded and laminated back on themselves, leaving conductors exposed on both sides of the fold (see figure 3). The folded conductors may then be soldered directly to the substrate with the aid of appropriate holding fixtures. This method is inherently low-profile, and can also produce extreme density when proper soldering techniques are used.

Lap soldering is an excellent way to produce dense connections in many flexible-circuit applications. When vapor-phase soldering is used with flex lap joints, mass termination of flex-circuit connections (many connections effected simultaneously) can be very economical. With good fixturing and in-line vapor-phase equipment, the process should be a simple sequence of applying solder paste to the PWB, baking the paste, cleaning and fixturing the parts, soldering, and post cleaning. Vapor-phase lap soldering has the potential of being able to terminate lines and spaces as small as 7 mils. This has actually been done on a small scale in a lab environment.

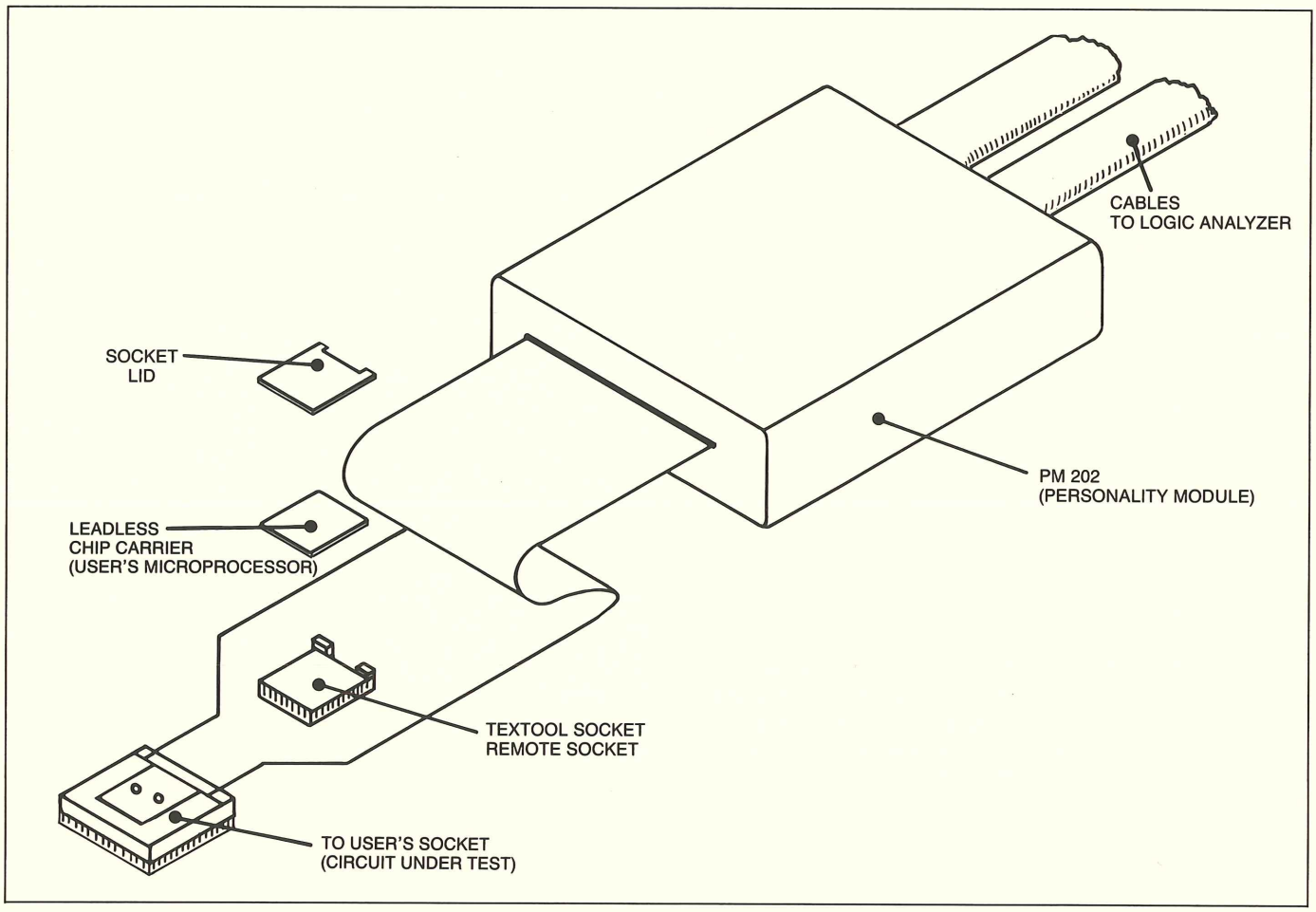


Figure 2. Problem: How do you connect a logic analyzer to a microprocessor in its actual operating environment? Imagine trying to attach 65 conventional probes to a functioning microprocessor in a chip carrier; attaching just one or two would be difficult, or even impossible. The solution: Put another socket (the remote socket) in parallel with the processor's normal home (the user's socket). The remote socket is mounted on and intergrated into the probe's circuitry. To do logic analysis, the probe is plugged into the user's socket and the processor to be tested is plugged into the remote socket. Flex circuits enable such probes to be more easily manipulated while preserving much of the processor's actual operating environment.

With the low-profile termination achieved, we needed to get the flex circuit to exit the user's socket through a .800'' × .050'' slot in the lid. How were we to route 68 conductors through that small opening? Fifty-one conductors had to be routed on one section of the flex, while the remaining 17 conductors would have to be routed on the other section. For 51 conductors to fit through the slot, the width of each of the runs and spaces had to be 7 mils.

Finer Lines Than PWBs – In general, flex circuits can have finer lines than conventional printed wiring boards, but minimum line widths and spaces vary from vendor to vendor. To assure good yields and consistent delivery, find out what your vendors can do before fixing your design. Ask them. Look at their literature.

Folding a Single Layer for Double Layer Performance

Next, we had to route all signals on one layer – cost goals dictated the use of a single-sided circuit to save money. After doing the routing, we folded part of the single-sided circuit over the first layer containing the remote socket and soldered this folded part to the remote-socket pins. The first layer of circuitry has the pattern appropriate for the component that will be placed at that location. The second, folded layer has the same pattern except the component-lead holes are drilled larger than the holes on the first layer. All exposed pads are then solder coated.

When the second layer of circuitry is folded and laminated onto the first, the large holes on the second layer line up to the first layer such that solder is exposed on both layers. During soldering, both layers will be soldered simultaneously (see figure 4). Folding and laminating achieves the functions of a double-sided circuit, using only single-sided materials.

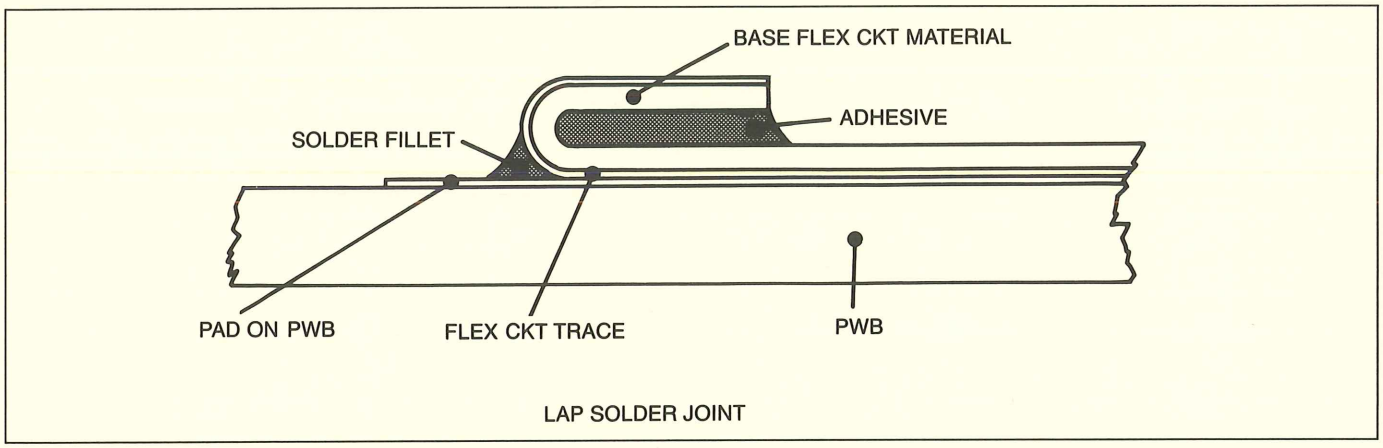


Figure 3. With dense flexible circuits, vapor-phase soldering is the best way to economically mass terminate many connections. (Mass terminate, in this sense, means connect simultaneously.) For soldering, the end of the flexible circuit is folded back and held by adhesive prior to soldering. In lab conditions, lines and spaces as small as 7 mils have been attached.

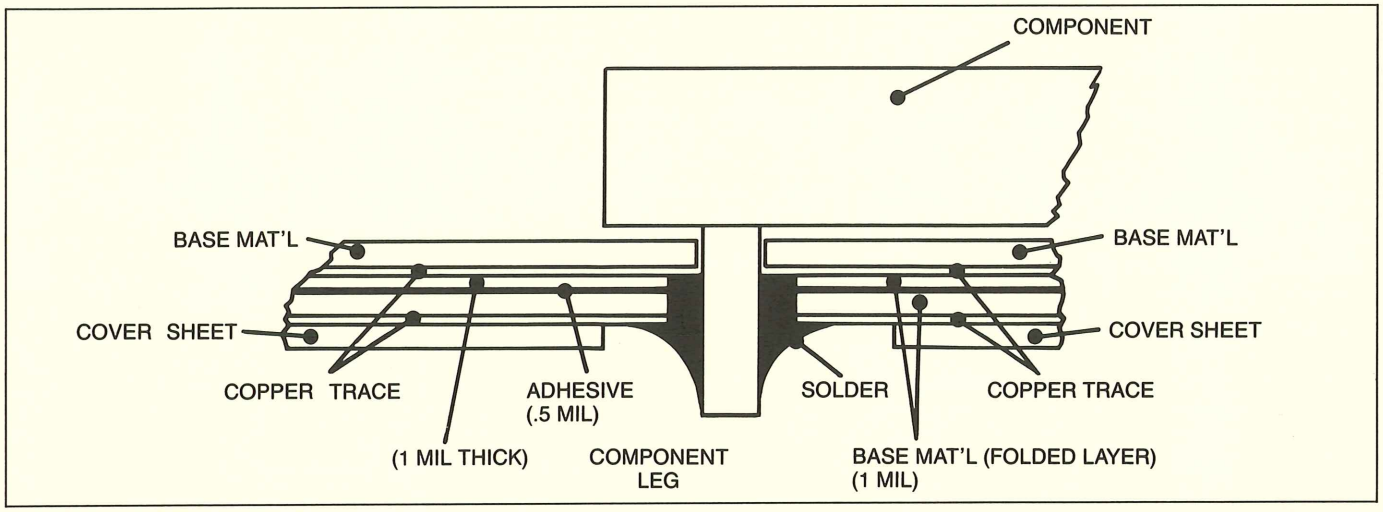


Figure 4. When folded, a single-sided flex circuit can approach double-sided performance. When holes are lined up using jigs, both layers can be simultaneously soldered.

There is significantly less cost in a single-sided circuit than in a double-sided circuit. Folded flex circuits offer great interconnect versatility and use less base material. For example, consider a flex circuit for routing signals inside an instrument. If such a circuit were laid out without folds, much base material in the flat could be wasted. If, instead, the layout was to done such as to minimize the amount of base material to be used, you might plan to fold the circuit to meet its final shape (see figure 5).

Base Material

The base material we chose for the probe application was copper-clad polyimide (Kapton). Kapton is very flexible, and was readily available in the desired thicknesses.

To resist fracturing, we specified the copper cladding to be *rolled annealed;* rolled annealed copper cladding can be flexed many times without failure. In contrast, electro-deposited copper cladding, although acceptable in static applications, tends to fracture when flexed repeatedly.

The Critical Path

Once all of the major design issues were resolved, it was time for our PWB design group (in DAG) to lay out the circuit for the first film work. They were to do layout from our drawings. They were also to order film and prototypes.

We were unpleasantly surprised to find lead times for prototypes in the flex-circuit industry (outside Tek) often are 10 to 12 weeks. For most projects this is much too long. We needed prototypes in no more than two weeks to meet project goals. Tek's Applied Chemical Components group (ACC) could meet our schedule.

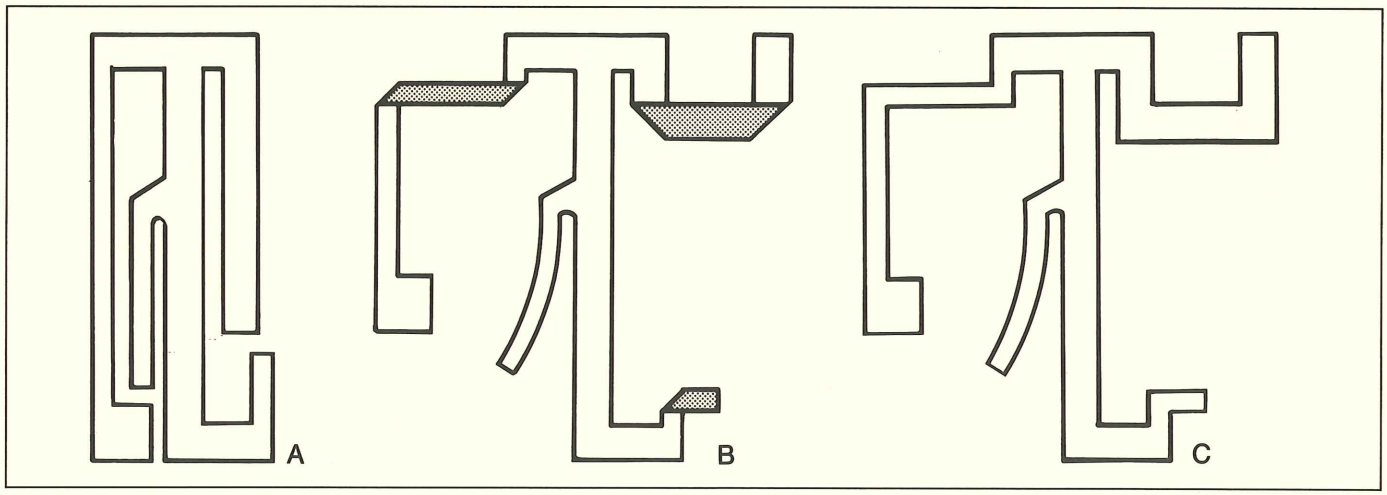


Figure 5. A little planning in layout can save considerable base material. Pattern A folds to produce finished circuit B, using less material than a no-fold pattern such as C.

The process of laying out the probe's flex circuit was more complex than expected. Our PWB designer didn't have flex-circuit layout guidelines. What's more, most PWB design guidelines don't apply to flexible circuits. We had to rely on vendors and their publications for the information. (See "Flexible Circuits, Part 2" for guidelines.)

After we ordered the first prototypes from Applied Chemical Components, our lack of flex-circuit information necessitated heavy communication with them. In these discussions, we found that early consultation with ACC or any other flex-circuit supplier is critical. This allows time for the supplier to study the circuit and suggest changes that can be most easily made before doing the first layout. Early consultation can save many film revisions and a lot of time and effort.

Ordering the Prototypes

When the film was ready, we assembled a package that included an order for ACC specifying materials and quantity, a film packet, and mechanical drawings (prints). During consultations, ACC helped us prepare these essentials.

The prints are critical and should include profile, circuit structure, special tolerances, plating and material call outs, hole sizes for both base material and cover sheet, and a detailed cross section of the circuit.

If the prototypes are going to be fabricated in-house or by any other source than the final manufacturer, it's a good idea to give that manufacturer time to come up to speed. Give the manufacturer time to build a "parallel" set of prototypes – We didn't! This oversight significantly delayed the shipment of the PM202 Personality Module for which the probe was an essential part. These delays could have been avoided by giving the vendor the chance to build prototypes *in parallel* with in-house prototyping. Even if you never use the vendor's prototypes, your money will be well invested.

The Prototypes Worked!

Despite the complex design, the first prototypes worked. Surprisingly, imaging and etching the fine lines proved to be much less of a problem than we expected. The problems that did show up related to inefficiencies in assembly and raw-circuit manufacturing. For example, we had to enlarge the cover-sheet opening to expose more conductor to solder to. (Cover sheets are equivalent to solder masks in PWBs.)

Other difficulties were cover-sheet misalignment, intermittent discontinuities, and solderability problems. The intermittents were the major problem in the first prototypes; they were caused by improper strain relief and the use of electro-deposited copper-clad material. Once we substituted rolled annealed copper, the discontinuities disappeared.

Alignment of lap solder joints before soldering is very important; it is also difficult without fixturing. With fixturing, soldering was easy. When your flexible circuit will require assembly, that is adding parts, design your fixtures early – before the first prototype assembly if possible. To use fixturing in the first build, the raw circuits must have tooling holes and features to match the fixtures. (We find it helpful to talk to flex-circuit vendors and in-house sources when planning fixturing.)

Shielding and Crosstalk

Late in probe development, crosstalk showed up as a major problem. Despite our earlier economic concerns, we now had to add a second circuit layer to eliminate the crosstalk between signal lines.

There are at least three basic ways to shield, not only against crosstalk but also against electromagnetic susceptence and radiation. You can add circuit ground-plane layers, screen on silver-bearing-ink layers, or vapor deposit aluminum layers on the circuit. Each method has advantages and disadvantages.

The choice of shielding method depends on balancing the electrical, mechanical, and environmental requirements. Adding one copper layer to the copper circuit as a ground plane, for example, will eliminate crosstalk, but suppressing electromagnetic radiation may require a layer on both sides. Unfortunately, adding *any layers to a flex circuit will degrade mechanical performance;* multilayer circuits are likely to have poor flex life.

Adding a second layer – the ground plane – lowers the flex performance significantly. The single-layer circuit tested to about 200,000 flexes; the double-sided circuit (signal lines over a ground plane) lasted only 2,000 cycles. To improve flex life, we imaged an array of holes in the copper layer that forms the ground plane. This increased flex life 50%, to about 3,100.

Single-sided circuits with a layer of silver ink screened on each side tend to do very well in flex-life tests. However, silver ink generally doesn't shield as well as copper. (Its conductivity, and therefore its shielding ability, depends on aspect ratio, thickness, and silver content.)

Then there is aluminum. I mention vapor-deposited aluminum only because the industry does use it. However, its effectiveness is questionable; it loses conductivity because it oxidizes quickly and becomes highly resistive.

The need for a ground-plane layer gave us another difficulty. How were we to electrically connect the two layers of circuitry (where connections to ground were needed) without plated-through holes? Plated-through holes both increase cost and reduce flex life – because the process adds electro-deposited materials to the rolled annealed copper. In our case the answer was easy.

The few points requiring feed throughs in our probe's flex circuit were in inconspicuous areas; therefore, eyeleting was acceptable. Wherever a feedthrough was needed, we placed an eyelet in a drilled hole and then soldered to pads on both sides.

Production Tooling

The tooling costs for flex circuits vary widely with volume and circuit complexity. For low volumes, a steel-rule die is sufficient for cutting out the profile; you will also need drill tapes, and folding, testing, and assembly fixtures. For low volumes, tooling costs are typically between $5000 and $10,000.

For high volume, you can't predict costs without specific circuit details. You will need a hardened-steel tool for cutting profiles. Therefore costs will certainly exceed those for low volume.

Conclusions

In the development of any flex-circuit design, especially those that are complex or unique, early delivery of prototypes and early planning are critical.

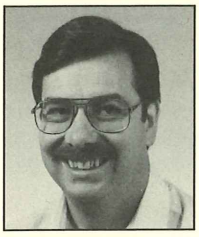
Easy, fast access to design information is also a key to success. If you don't have this information, see your vendor. In any case, establish vendor contact early to confirm schedules and manufacturability. Without considering these factors, a flex circuit can become the critical-path item and drive your project schedule out weeks.

For More Information

For more information, call David Payne 629-1887 (92-716). □

Flexible Circuits Part II

GETTING MORE OUT OF YOUR FLEX-CIRCUIT DESIGN



Gary Uchytil is a senior electrochemical process engineer in Applied Chemical Components, part of EMCO. Gary joined Tek in 1976 from Plessy Micro Science, where he was engineering manager. He is a past president of the Photo Chemical Machining Institute and a past vice president of the California Circuits Association. Gary has a BS in general science from Oregon State University.

The logic analyzer probe (PM202) described by Dave Payne in "Flex Circuits Answer the Challenge of a Complex-Probe Design" was one of the first flex circuits made in Applied Chemical Component's Flex Circuit business element. In this article, the author relates the process of developing the probe in the context of what the division-level designer can do to achieve more effective flex-circuit designs.

Applied Chemical Components (ACC) is now producing electrically tested flex-circuit assemblies complete with soldered-in components. Many of these assemblies use suface-mounted devices (SMDs). Many are vapor reflow soldered. We sell these assemblies to the divisions as part of larger assemblies, acting as a general contractor to the divisions.

Manufacturing in ACC is uniquely organized into business elements. Each business element has the complete manufacturing responsibility for a product or products – from start to finish. The products of a business element share a common technology, usually implied by the organization's name: Photo Etch Products, EMI Shielded Products, High Frequency Products, Stainless Steel Etch Products, etc. These business elements are equipped to do work requiring tight tolerances: ±0.0005", 0.003" lines and spaces, 0.001" thick material, and registrations as close as 0.0002".

When division designers began to ask for flexible circuitry, we formed a new business element, Flexible Circuit Assemblies, and used the technology we had developed in the Photo Chemical Machining group. The Flex Circuit Assemblies business element can produce prototype flex circuits and assemblies fast – within 10 days. They can also handle production runs.

The Challenge of the Emulator Probe

Logic Analyzer engineering contacted the Flex Circuit business element after they had completed the probe's physical layout and needed a prototype. Because of the probe's complexity and novel features, the LA team was uncertain about how manufacturable the probe would be. We couldn't answer all their questions immediately, but we worked out answers as we made the first prototype. The major uncertainties were:

- Would copper take the tight bend required without cracking? The flex circuit required the copper-clad polyimide to be tightly folded (creased). We found nothing describing this procedure in the literature.
- Because a LCC socket had to be connected to two circuit layers, would the solder wick down the leads to the second circuit?
- Could we go beyond the tolerances of conventional pin registration? Pad sizes were limited because the probe design called for three runs to go between pads. These runs were to be 0.006" wide with 0.004" spacing. This made the coversheet registration difficult.
- Would material wastage increase the price of the end-product of which the flex circuit was a part – perhaps beyond the targeted price? Because the circuit was large, only two circuits could be printed on one sheet of the base material.

The key to the probe project's success was ACC's early participation in the design. If ACC had not known the requirements before the prototype was to be built, the project would have been delayed. Both Logic Analyzer engineering and ACC agreed that the prototype delivery had to be reduced from the 12 weeks quoted by outside flex vendors. We delivered in four days, six days sooner than the 10-day target. Early delivery helped Logic Analyzers meet its schedule – even after making modifications based on evaluations of the prototype's performance.

Design Rules

This part of the article presents 14 design rules. These rules complement the flex-circuit manufacturing process. I've related them to the logic analyzer probe (PM202) to clarify their value to the division-level designer.

Rule 1

The PM202 Probe required a sharp bend to complete the circuit pattern.

- Design rule 1: In any bend area, the runs should be as perpendicular to the bend as possible. (See figure 1.)

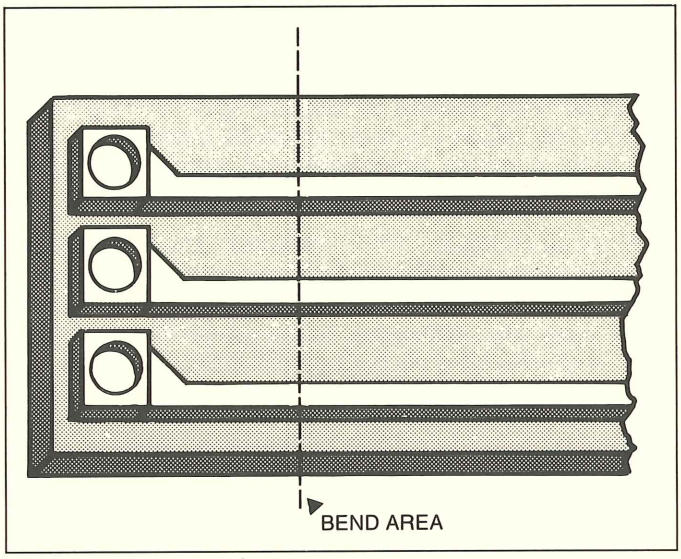


Figure 1. Runs should be perpendicular or nearly perpendicular to a bend point or area.

Rules 2 and 3

The polyimide that is the base material for flex circuits can be purchased with either electrodeposited or rolled annealed copper cladding. The copper is fastened to the polyimide with either modified epoxy or acrylic adhesives. After repeated foldings, the rolled annealed copper cladding and acrylic adhesive used for the PM202 Probe didn't crack during bend testing; this combination proved superior, with the longest test life.

- Design rule 2: When the circuit must be sharply bent or the circuit will be flexed more than 1000 times in actual use, specify rolled annealed copper clad with acrylic adhesive.
- Design rule 3: In dynamic areas (where flexing will occur), runs should not take 90-degree angular turns. If turns must be made, always lay out smooth curves. (See figure 2.)

Rules 4 through 8

Photo Chemical Machining's knowledge of processing thin material and rolled-metal stock was invaluable in producing the prototype flex circuits for the probe. And because chemical processing does not mechanically damage the surface of the material, our yields are 95% and greater.

At Tektronix, holes in copper-clad polyimide are NC drilled. We also NC drill the polyimide cover sheet to expose circuit pads for soldering. Most outside vendors stamp holes in the polyimide with hard dies.

- Design rule 4: At Tek, square pads rather than round pads are preferred because the circuit will be NC drilled. Runs should attach at the corner of the pads, as shown in figure 3.
- Design rule 5: In all cases, square pads are mandatory when the runs are 0.010" wide or narrower.
- Design rule 6: Rolled annealed copper has a grain. Runs that follow the direction of this grain will survive more bending. On your drawings, always specify the area that will flex and the grain direction needed to maximize flex life.

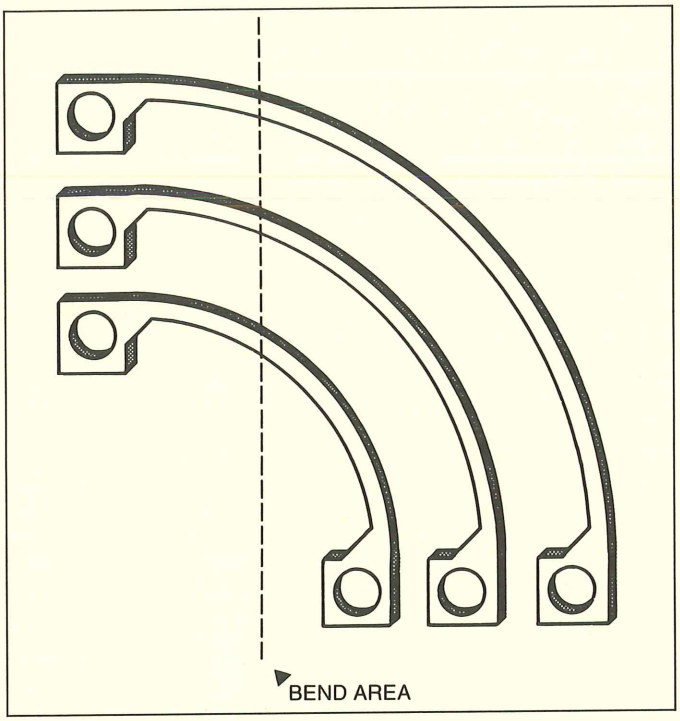


Figure 2. 90-degee angular turns should not be placed in areas where repeated flexing will occur. Use smooth turns instead.

- Design rule 7: Always specify 1 oz. copper on your drawing.
- Design rule 8: Always highlight critical tolerances on your drawing.

Rule 9

Flex-circuits are made by photo etching. A photosensitive coating is applied to the copper cladding; this coating is exposed to an image created by the film work, developed, and the exposed copper is etched away. This process is done in a clean room because one tiny dust particle can be fatal, causing a reject. The flex-circuit process can yield 0.003" lines and spaces (see figure 4). The imaging process is exact enough for 0.0005" front-to-back registration of circuitry.

The film tooling is generated by a CAD system and photo plotted by ACC's Graphic Tooling group. The circuits are plotted onto film or glass. In either case, we keep the master within ACC to prevent damage.

- Design rule 9: Show critical line widths and spacings on your drawing. If line-width minimums are critical, we can compensate for the etching process by adding width to the photo tooling.

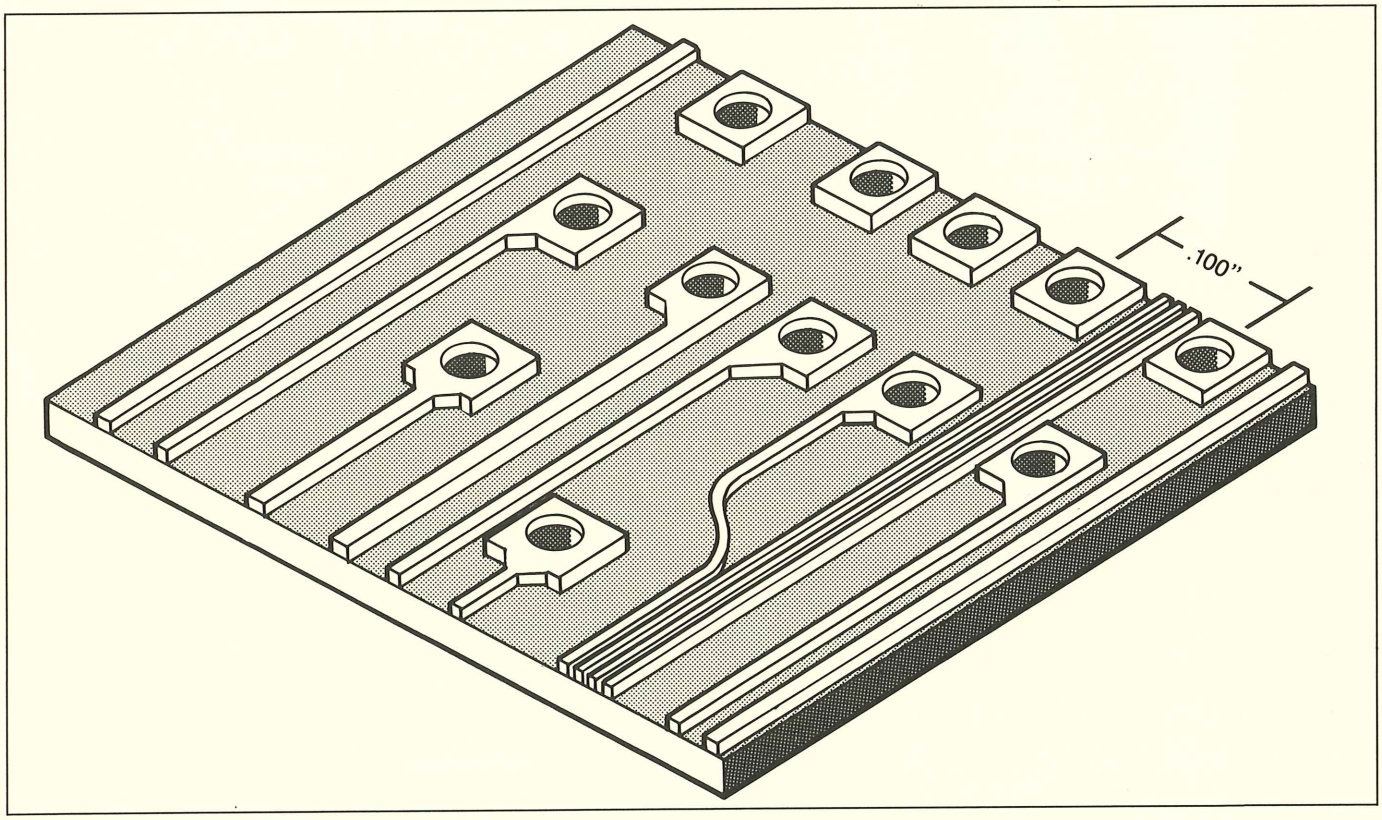


Figure 3. Runs should be attached to a pad at the pad's corner.

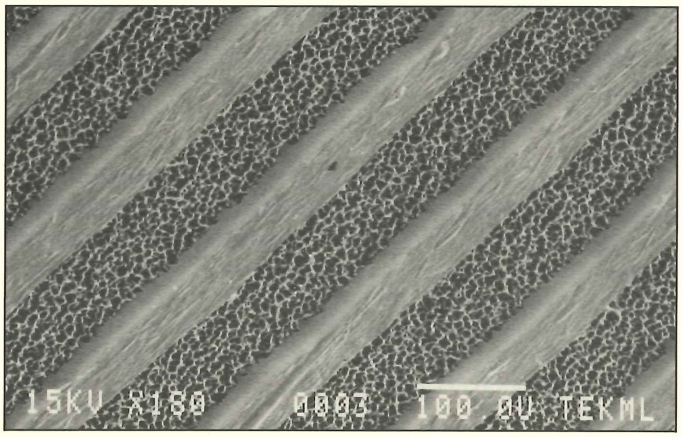


Figure 4. Lines on ACC-made flex circuits can be as narrow as 0.003". This SEM photo was taken using 180 times magnification.

Rules 10 and 11

After etching, the copper runs are covered with a polyimide sheet of the same thickness as the base polyimide; this cover sheet acts much like a solder mask on a PWB. The cover sheet is durable and protects the copper runs from damage, shorts, and wear. The pads or other areas to be soldered are coated with solder or immersion tin.

- Design rule 10: The cover-sheet holes should be 0.015" larger than the diameter of the drilled hole.
- Design rule 11: If a group of pads are to be bared (exposed) by a window in the cover sheet, then the registration tolerance of the window will be ±0.50". WARNING: A window cut in the cover sheet to bare pads can create a natural bend point that will, with repeated bending, break conductors at the edge of the cover sheet window.

Rules 12 and 13

The flex circuit is then stamped out with a steel-rule die, inspected, and continuity tested. Next, the circuit is sent to assembly for component mounting, soldering, and electrical testing.

- Design rule 12: Set aside a 0.75" area outside the circuit itself to allow for steel-rule-die tooling holes.
- Design rule 13: Make component mounting holes 0.010" larger than the lead to be inserted.

Vapor reflow soldering

In earlier PM202s, the small PWB was lap soldered by hand. Today, we blind solder this PWB by applying solder paste and using vapor reflow soldering. This improves yields and productivity.

ACC's Other Flex-Circuit Capabilities

The PM202 was the first Tek probe to employ a flex design. Because LA engineering and ACC flex circuit engineering were closely coupled, the PM202 met a tight marketing-set schedule.

Although the PM202 was challenging, it didn't require all the processes we have available. We can also give you plated-through holes, "rigidized" flexible circuitry, and vias.

Plated-through holes

The polyimide base for flex circuits can support plated-through holes for connecting runs on opposite sides. When using plated-through holes, we plate only the pad and hole areas. The electrodeposited copper must be kept off the runs or flexing may cause cracking. (With plated-through holes, pads are about 0.0015" higher than the runs.)

Rigidizing flexible circuitry

Areas of a flex circuit can be reinforced, or "rigidized" with a stiffener, FR-4 for example. These areas are usually used for mounting components or connectors. The rigidized area is usually a two-sided circuit on polyimide, laminated to a stiffener having matching hole locations. The holes in the stiffener are not plated through; in fact, there is usually no circuitry on the stiffened area (other than mounted components and connecting runs).

- Design rule 14: Make the holes in the stiffener 0.020" larger than the holes in the flex circuit.

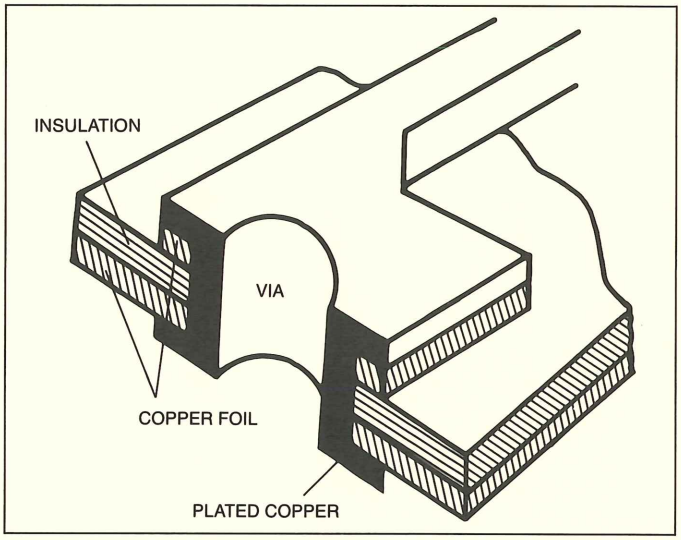


Figure 5. Vias in flex circuits can be as small as 0.010" in diameter. Vias are used to electically connect runs on opposite sides of flex circuits.

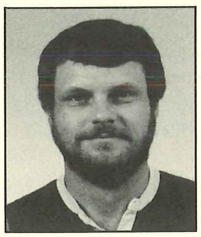
Vias

To electrically connect runs on both sides of the polyimide, very small holes can be NC drilled in the copper-clad polyimide and through-hole plated. These small holes are called vias. Vias in flex circuits can be as small as 0.010" in diameter. Many surface-mounted-device circuits are etched on copper-clad polyimide. To save space, the electrical connection between the circuits on the front and on the back are made with vias drilled in the mounting pad.

For More Information

This article has discussed some of the features used in flex circuits and has included a manufacturing guide. By no means have all applications been detailed, or even mentioned. To get more information on other applications and to discuss special designs, contact Gary Uchytil, at 627-0321 (32-330). □

MIXING LANGUAGES ON UNIX



Joel Swank is software engineer in the Software Center, part of Software Tools Support. Joel joined Tek in 1975. Earlier he was a systems programmer for the State of Virginia. Joel has a BS in computer science from Virginia Tech.

Although UNIX provides a variety of high-level language compilers, UNIX languages are not designed to work together. This is unfortunate but not an insurmountable barrier. This article shows how three of the most popular languages on Unix — C, Fortran, and Pascal — can be mixed.

Trying to link routines from one language with routines in another can be frustrating, but by following a few guidelines, such languages can be successfully mixed, giving programmers access to many powerful subroutine libraries. Mixing also allows sections of large applications to be written in the best language for the task.

In this article, we will examine how C, Fortran, and Pascal can be mixed. I have included 15 working examples developed by the Software Center.

The example programs were developed under UNIX 4.1BSD, TEK version 1.1. We have also tested the programs under UNIX 4.2BSD. We found all 4.1BSD concepts work equally as well under UNIX 4.2, except Fortran output to stdout.

The big problem in mixing two languages is determining what data types are equivalent. Some data types have no direct equivalents but can be emulated; for example, only Fortran has complex variables, but an equivalent C structure can be defined. Other data types have no equivalents; for example, Pascal has no single-precision floating-point numbers.

Mixing C and Fortran

UNIX 4.2BSD Fortran

There is a slight difference in the way 4.2 Fortran and 4.1 Fortran handle *stdout.* When stdout is used in a Fortran subroutine called from C under UNIX 4.2BSD, the output is sent to a file called *fort.6* instead of *stdout.* You can circumvent this bug/feature by using the symbolic-link feature of UNIX 4.2 to reassign the output. To direct output to the terminal, use the command *ln -s /dev/tty fort.6* before running the program; this circumvention prevents the shell from redirecting the Fortran output. An alternate method allows normal use of stdout. You create a dummy Fortran main program that does nothing except call the real mainline C program and return.

Equivalent data types

Here are the equivalent data types for Fortran and C:

FORTRAN	C
integer*2 x	short int x
integer x	long int x
logical x	long int x
real x	float x
double precision x	double x
real*8 x	double x
complex x	struct { float r, i} x
double complex x	struct { double r, i } x
character*10 x	char x[10]

Fortran and C character strings are not exactly equivalent. A null character (binary 0) marks the end of a C string. A Fortran string must be padded with nulls to the defined length of the string. In all other respects, the character strings are equivalent.

Naming conventions

The Fortran compiler appends an underscore (_) to all external identifiers. This means that Fortran programs can call only C routines named with a trailing underscore. It also means Ftn. subroutines must be identified in C routines by appending an underscore to the name of the Fortran routine.

Passing parameters

C normally passes parameters by value; that is, the values of the parameters are put on the stack. Fortran passes parameters by address; that is, pointers to the values are put on the stack. You can overcome this incompatibility by forcing C to pass by address. A C calling routine must specify parameters to be passed to Fortran using the 'address of' notation. For example: *fsub_(&n,&x);*

A C routine called from Fortran must receive pointers as parameters. The parameters must be declared as pointers using the 'pointer to' notation. (For example: *float *n, *x*) This need to specifiy pointers is not true of arrays in C. In C, arrays are always passed by address.

Multi-dimensional arrays

C and Fortran also differ in the way each stores multidimensional arrays. In C, the rightmost subscript varies fastest; in Fortran, the leftmost subscript varies fastest. You can overcome this problem by declaring the subscripts in the opposite order in the two languages. For example a Fortran array such as *real x(5,4)* would be declared as *float x[4][5];* in C.

C and Fortran also index array elements differently. C numbers the elements of an array of n elements from 0 to n-1. In the same array, Fortran numbers the elements from 1 to n. If indices are passed, 1 must be added to each index passed from C to Fortran, and 1 must be subtracted from each index passed from Fortran to C.

Routines

Fortran has two types of routines: the *function* and the *subroutine.* C has only one: the *function.* But by usage, the programmer can make a C *function* equivalent to either a Fortran *function* or a Fortran *subroutine.* A Fortran function differs from a Fortran subroutine in that it returns a value to the calling routine while a Fortran subroutine does not. For a Fortran routine to be callable as a function, that routine must be declared a function in its first line. Also, the value to be returned must be assigned to the function name.

For a C routine to be callable as a function, the routine must specify the return value on the return statement. The way a C routine is called depends on whether it is a function or a subroutine. In Fortran, subroutines must be called using the call statement; a function is called by using it as a value in an assignment statement. In C, a subroutine is called by specifying its name and parameters as a complete statement and a function is called by using it as a value in a statement.

Linking

To link Fortran and C programs, special linkage editor commands are needed. For example:

Normal C to F77 link:
%ld –X /lib/crt0.o (your objects) (your libs) –l77 –lF77 –lm –lc

If the –p option was used on any of the compiles:
%ld –X /lib/mcrt0.o (your objects) (your libs) –l77 –lF77 –lm –lc

If the –g option was used on any of the compiles:
%ld –X /lib/crt0.o (your objects) (your libs) –lg –l77 –lF77 –lm –lc

Examples 1 through 4 show C routines calling Fortran routines. Examples 5 through 7 show Fortran routines calling C routines.

Mixing C and Pascal

Data types

Pascal has fewer data types than either Fortran or C. Although all Pascal data types have at least a rough equivalent in C, not all C data types have a Pascal equivalent. The Pascal data type *Boolean* has no direct equivalent in C. A Pascal Boolean data type can be interpreted as a C character data type, but the only values valid are binary 0 (false) and binary 1 (true). Here are the equivalent data types:

Pascal	C
var x: integer	long int C
var x: real	double x
var x: packed array[1..10] of char	char x[10]
var x: boolean	char x

Passing parameters

Pascal normally passes all parameters by value. Except for arrays, C does too. To pass arrays to and from C, the array must be declared in the Pascal function or procedure definition with a *var* statement. The use of *var* in a definition causes all parameters to be passed by address.

Pascal arrays are stored in memory in the same way as C arrays. A Pascal array of n elements is indexed from 1 to n. A C array of n elements is indexed from 0 to n-1. This means indices passed from Pascal to C must have one subtracted from their values, and indices passed from C to Pascal must have one added to their values.

Parameters other than arrays can also be passed by address by using the *var* statement in the Pascal procedure definition. For example: *procedure psub (var x,y:real);*

The C routine must also pass by address. If C is the calling routine, the parameters must be coded using the 'address of' notation. For example: *psub (&x,&y);*

If C is the called routine, the parameters must be declared as pointers. For example: *float *x, *y;* Declaring variables as pointers allows the values of parameters to be updated directly by the called routine.

Routines

Pascal has two types of routines, the *function* and the *procedure.* C has only one, the *function.* But by usage, the programmer can make the C *function* equivalent to either the Pascal *function* or the Pascal *procedure.* A Pascal function differs from a Pascal procedure by returning a value to the calling routine; a Pascal procedure does not. For a Pascal routine to be callable as a function, it must be declared as a function in the first line of the routine and the data type of its return value must be specified. Also the value to be returned must be assigned to the function name.

For a C *routine* to be callable as a *function,* it must specify the return value on the return statement. The way a routine is called also depends on whether it is a function or a procedure. In both Pascal and C, procedures must be called by using the function name and parameters as a complete statement; a function is called by using it as a value in an assignment statement.

Linking

To link Pascal and C programs, special linkage editor commands are needed. For example:

Normal C to Pascal link:
%1d –X /lib/crt0.o (your objects) (your libs) –lpc –lc

If the –p option was used on any of the compiles:
%1d –X /lib/mcrt0.o (your objects) (your libs) –lpc –lc

If the –g option was used on any of the compiles:
%ld –X /lib/crt0.o (your objects) (your libs) –lg –lpc –lc

Pascal include files

Standard Pascal requires that all *function* and *procedure* definitions be included from an 'h' file rather than coded in-line. This is not enforced in the current version of the 4.1 bsd compiler. For simplicity, my examples violate this restriction. In practice, all procedure and function definitions should be in separate 'h' files to maintain portability and upward compatibility to new releases of the compiler.

Examples 8 through 11 show C routines that call Pascal routines. Examples 12 through 15 show Pascal routines that call C routines.

Mixing Pascal and Fortran

Because Pascal will not accept the underscore (_) character in identifiers, Pascal and Fortran cannot call each other directly. But if you use a C interface routine, this restriction is bypassed and Pascal and Fortran can be mixed.

For More Information

For more information, call Joel Swank, 627-4403 (50-487). □

Fifteen Working Examples

EXAMPLE 1. C calling a Fortran function as a function.

```
/*    Example of calling a function, called fsqrt, written in fortran
                from a C program
*/
main()
{
   float x,y,fsqrt_();   /* declare fsqrt as a fortran real function */
   x=45;
   y=fsqrt_(&x); /* pass the address of x */
   printf("the square root of %f is %f\n", x,y);
}
C    This fortran function can be called as a C function
C
      real function fsqrt(x)
      real x
C     return value is assigned to the function name
      fsqrt=sqrt(x)
      return
      end
```

EXAMPLE 2. C calling Fortran passing a 2-dimensional array.

```
/*  Example of calling a fortran subroutine from C , passing a 2
            dimensional array    */
/*
*/
main()
{
   float x[5][4];
   int i,j;
   /* fill the array */
   for (i=0; i<5; i++)
   /* subscripts vary from 0 to n-1 */
   {  for (j=0; j<4; j++)
            {
            /* rightmost subscript varies most rapidly */
            x[i][j] = i*10+j;
            printf(" %f",x[i][j]);
            }
            printf ("\n");
   }
   /* pass the array to fortran subroutine for printing */
   out_(x); /* arrays are always passed by address */
}

C    Fortran subroutine that receives a 2 dimensional array from
C          a main program written in C
C
      subroutine out(x)
C     subscripts are in the opposite order
      real x(4,5)
C     index goes from 1 to n
      do 10 i=1, 5
C     leftmost subscript varies most rapidly.
  10  write (6, 20) (x(j,i),j=1,4)
  20  format (6f5)
      return
      end
```

EXAMPLE 3. C calling Fortran passing character strings.

```
/*    Example of a C routine that calls a fortran subroutine
            passing two character strings.
*/
#define MAXL  80
#define EOF 5
main()
{
   int len;
   char line[MAXL],line2[MAXL];
   len=getline(line,MAXL);
   len=getline(line2,MAXL);
   char_(line,line2);
}
getline(s, lim) /* get line into s, return length */
char s[];
int lim;
{
   int c,i;
   for (i=0; i< lim; ++i) s[i]='\0' /* clear string for fortran */
   for (i=0; i< lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
      s[i]=c;
   return(i);
}
```

```
C    Subroutine that recieves two character strings from a C calling
C          routine, concatenates them and prints the result.
C
      subroutine char(line,line2)
C    parameters longer than 40 will be truncated
      character*40 line,line2
      character*80 line3
      line3=line//line2
      write(6,30) line3
   30 format (a80)
      return
      end
```

EXAMPLE 4. C calling Fortran to perform complex number calculations.

```
/*    Example of a C routine that calls a fortran subroutine to
          perform calculations on complex numbers
*/
main()
{
/*  define a complex data type to C     */
   struct complex { float r; float i ; } ;
   struct complex m,n;
   m.r= -81;
   m.i=15;
   n.r= -81;
   n.i=15;
   cxrt_(&m); /* pass address - value returned in m   */
   printf("The square root of %f,%f is %f,%f\n",n.r,n.i,m.r,m.i);
}
```

(Example 4 continues next page)

(Example 4 continued)

```
C   Example of a subroutine called from C to do complex calculations
C
      subroutine cxrt(num)
      complex num
      write (6,30) num
      num=sqrt(num)
      write (6,30) num
      write (6,30) num*num
   30 format (6f10.5)
      return
      end
```

EXAMPLE 5. Fortran calling C routine as a function.

```
C     f77 routine that calls C routine as function
C
C   Declare the external c function
      external  csqrt
C   Both parameter and return value data type must match
C        data types in declared in the C function.
      real*8 b,r
      b=81
      r=csqrt(b)
      write (6,30) b,r
   30 format('the square root of ',f10.5,' is ',f10.5)
      end
```

```
/*  C function called from f77 as a function */

/*  data type of return value of function must be declared  */
double csqrt_(x)
/*   data type of parameter declared */
double *x;
{
   double y,sqrt();
   y=sqrt(*x);
   printf("the square root of %f is %f\n",*x,y);
/*  value to be returned must be on return statement, and data type must
        match the data type declared for the function */
   return(y);
}
```

EXAMPLE 6. Fortran calling C passing and returning parameters.

```
C
C —    test of f77 calling C passing 2 reals and an array
C
      dimension arry(11)
C   program will add first 10 array elements
      data cnt/10/
      data arry/1,2,3,4,5,6,7,8,9,10,11/
      call addum(cnt,arry,tot)
      print *,tot
      end
```

```
/*
    C subroutine designed for calling from F77
*/
addum_(cnt,arry,tot)
/* all parameters are pointers  */
float *cnt, *tot, *arry;
{
   float i;
   *tot=0;
   for (i=1; i < = *cnt; i++) {
      *tot = *tot + *arry;
      arry++;
   }
}
```

EXAMPLE 7. Fortran calling C passing a 2-dimensional array.

```
C    Example of a fortran routine routine passing a
C          2 dimensional array to C
C
      real x(4,5)
C   leftmost subscript varies most rapidly
      do 10 i=1,5
      do 10 j=1,4
   10 x(j,i)=j*10+i
      do 20 i=1,5
   20 write (6,30) (x(j,i),j=1,4)
   30 format (6f4)
      call carray(x)
      end
```

```
/*  C routine that receives a 2 dimensional array from Fortran
*/
carray_(x)
/*   subscripts are reversed    */
float x[5][4];
{
   int i,j;
/* rightmost subscript varies most rapidly - indices start at 0   */
   for (i=0; i< 5; i++) {
      for (j=0; j< 4; j++) {
      printf(" %f",x[i][j]);
      }
      printf("\n");
   }
}
```

EXAMPLE 8. C calling a Pascal function.

```
/*        Example of C calling Pascal function
*/
main()
{
 /*  double in C equals real in Pascal  */
   double n,m,psub1();
   n=22;
   m=psub1(n);
   printf("%f is the square of %f\n",m,n);
}
```

```
(*  Pascal function to be called as a C function  *)

function psub1(x:real):real;
begin
  psub1 := x*x;
end;
```

EXAMPLE 9. C calling Pascal procedure passing parameters both ways.

```
/*  Example of a C routine that calls a Pascal subroutine, passing
            parameters both ways */
main()
{
   double n,m,psub2();
   n=22;
   /*  parms passed as addresses so Pascal can modify them */
   psub2(&n,&m);
   printf("%f is the square of %f\n",m,n);
}
```

```
(*  Parameters declared with var because addresses are
       being passed from C *)

procedure psub2(var x,y:real);
begin
   y := x*x
end;
```

EXAMPLE 10. C calling Pascal passing character strings.

```
/*   Example of a C routine that calls a Pascal subroutine
              passing two character strings.
*/
#define MAXL  80
#define EOF 5
main()
{
   int len;
   char line[MAXL],line2[MAXL];
   printf ("enter line 1: ");
   len=getline(line,MAXL);
   printf ("enter line 2: ");
   len=getline(line2,MAXL);
   psub4(line,line2);
}
getline(s, lim) /* get line into s, return length */
char s[];
int lim;
{
   int c,i;
   for (i=0; i< lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
      s[i]=c;
   s[i] = '\0';
   return(i);
}
```

```
(* Example of a Pascal routine that receives strings from C  *)

(*  declare a string as a packed array of char  *)
type string = packed array[1..80] of char;

(*  declare procedure with var because C passes address of strings *)
procedure psub4(var st1,st2:string);
begin
   writeln(st1,st2);
end;
```

EXAMPLE 11. C calling Pascal passing a 2 dimensional array.

```
/*   Example of calling a Pascal subroutine from C , passing a 2
              dimensional array    */
/*
*/
main()
{
   double x[5][4];
   int i,j,y,psub5();
   for (i=0; i< 5; i++)
   {  for (j=0; j< 4; j++)
            {
            x[i][j] = i*10+j;
            printf(" %f",x[i][j]);
            }
            printf ("\n");
   }
   psub5(x);
}
```

```
(*  Pascal routine that receives a 2 dimensional array from C  *)

type myarray = array[1..5,1..4] of real;
(* array to be passed must be declared in a var statement
          because a pointer is passed  *)
procedure psub5(var x : myarray);
   var i,j:integer;
begin
   for j := 1 to 5 do
   begin
      for i := 1 to 4 do
      begin
      write(' ',x[j,i]:5:1);
      end;
      writeln;
   end;
end;
```

EXAMPLE 12. Pascal calling C as a procedure.

```
(*  Example of Pascal calling C as a procedure  *)

program test2(input,output);
var x,y:real;
procedure csub2(x,y:real);external;
begin
  x := 6.9e15;
  y := 42;
  csub2(x,y);
end.
```

```
/* C routine that receives two doubles from Pascal   */

csub2(x,y)
double x,y;
{
   printf("%f, %f\n",x,y);
}
```

EXAMPLE 13. Pascal calling C as a function.

```
(*   Example of Pascal calling a C subroutine as a function  *)

program test4(input,output);
var x,y:real;
(*  C subroutine declared as a Pascal function   *)
function csub4(x:real):real;external;
begin
  x := 8130673.61536374653;
  y := csub4(x);
  writeln('the square root of ',x,' is ',y);
end.
```

```
/* Example of a c routine that is called as a Pascal function  */

csub4(x)
/*   reals in Pascal are doubles in C   */
double x;
{
   double y;
   y=sqrt(x);
   return(y);  /* must include return value on return statement */
}
```

EXAMPLE 14. Pascal calling C passing a 2-dimensional array.

```
(*   Example of pascal passing a 2 dimensional array to C   *)

program test6(input,output);
type myarry = array[1..5,1..4] of real;
var x: myarry;
   i,j:integer;
procedure csub6(var x: myarry);external;
begin
   for j := 1 to 5 do
   begin
      for i := 1 to 4 do
      begin
      x[j,i] := j*10+i;
      write(' ',x[j,i]:5:1);
      end;
      writeln;
   end;
   csub6(x);
end.
```

(Example 14 continues next page)

(Example 14 continued)

```
/* C routine that receives a 2 dimensional array from Pascal

csub6(x)
double x[5][4];
{
  int i,j;
  for (j=0; j<5; j++) {
    for (i=0; i<4; i++) {
    printf(" %f",x[j][i]);
    }
    printf("\n");
  }
}
```

EXAMPLE 15. Pascal calling C passing a character string.

```
(* Example of Pascal passing a string to a C subroutine *)

program test8(input,output);
(* declare a string as a packed array of char *)
type string = packed array[1..80] of char;
var st : string;
   ct : integer;
procedure csub8(var st: string);external;
begin
     write('enter string: ');
     for ct := 1 to 80 do
          if not eoln and not eof then read(st[ct])
          else st[ct] := chr(0); (* C requires the null terminator *)
          readln;
     csub8(st);
end.
```

```
/* C routine that receives a character string from Pascal */

csub8(st)
char *st;
{
  printf("%s\n",st);
}
```

AN UPDATE ON TEK's SOFTWARE TOOLS PROGRAM

Chuck Martiny is the manager of the Software Center, part of the Computer Science Center. Chuck joined Tek in 1976 from the Computer Sciences Corporation with more than 20 years in software development and other software-related experience. He holds a BSEE from the University of Arizona.

The Software Tools Program is almost one year old. Already, parts of the program are adding to the knowledge and productivity of engineers throughout Tek. Other parts are still developing – standardizing on a UNIX version for Tektronix, for example. This article reports on who is doing what and details the help and training now available to Tek software engineers and software users.

The Software Tools Program was developed in the Technology Group's Software Center. Formed in 1980, the Center helps improve the productivity of software engineers and the quality of software developed by the product groups at Tektronix. (Software Center services are described in the sidebar.)

The Software Tools Program Today

In 1982, the Software Center surveyed software users at Tek. After evaluating user responses to this annual survey, we decided two areas needed extensive study:

(1) The software tools that Tek and the industry are using

(2) The tools Tek will need for the greatest productivity in the future

After Norm Kerth (now in GPP) conducted the study, we concluded that the Center should concentrate on UNIX as an operating environment and that Tek needed to concentrate on seven areas within UNIX:

- A Tek-standard UNIX
- Production-quality compilers
- A configuration-management system
- A high-level-language debugger
- Specification and design tools
- Software-test tools

Based on these conclusions, we have developed a proposal as part of the Software Tools Program. Our study is detailed in a report that served as the justification for the Tools Program. The report stressed Tek's need to standardize on a few high-quality tools. With these standard tools:

- Software engineers could share information and programs, reducing product-development time.
- Spend less time and money maintaining the many sets of tools now in use.
- Improve the quality of Tek products – with less effort and cost – by using proven tools and software modules.

Seven Groups Work for the Success of the Tools Program

For efficiency, the Tools Program was divided into six parts, giving most program responsibilities to talent within five existing and two new Tek organizations: The UNIX Steering Committee (new), the UNIX Training group, the Tools Support group (new), the UNIX Documentation group, the UNIX User Assistance group, and the UNIX System Support group. These six operations, plus the Software Center, compose the Software Tools Program.

Although the Software Center was the driving force behind the program and continues to coordinate many of the activities, the Tools Support group is the only part reporting into the SWC.

UNIX Steering Committee

The Unix Steering Committee works to assure that Tektronix uses a single, standardized version of UNIX, and that this version is well supported throughout the company. The steering committee also recommends and establishes corporate UNIX strategies and works to resolve issues requiring agreement by the operating divisions.

Tek's current UNIX standard is an enhanced version of Berkeley 4.2. The Committee is defining the version for the next release. In this consideration, the primary issues are the release's compatibility for development, its match to future product requirements, and its conformance with national standards.

For more information about UNIX Steering Committee activities, contact chairman Jon Marshall at 685-2233 or Chuck Martiny at 627-6834.

UNIX training

Technical Communications (in the Scientific Computer Center, managed by Carolyn Schloetel) provides UNIX training as support for those doing product development. This is on-the-job training for programmers, design engineers, and manufacturing and support personnel.

Three staff instructors have developed and are teaching classes for novice users (introduction, files and directories, mail, and the vi and ms commands) and intermediate users (Systems Workshop). Other classes are being developed for shell programming, 4.2 internals, device drivers, and UNIX tools.

Videodisc courses on UNIX fundamentals and C programming are offered. Two interactive videodisc systems are available for student use by rental from Technical Communications. Because this computer-based instruction is interactive, users can learn at their own speed or address particular areas of UNIX. For details call Carolyn Schloetel, 627-1762.

THE SOFTWARE CENTER

Among the services the Software Center provides are:

Tools Support – The Center researchs, evaluates, selects, and obtains major software packages to help engineers with their software tool needs.

Software Center Information Exchange – A collection of example and reference documents for software engineers and project managers. It includes more than two hundred documents in 21 categories covering topics such as post-project reviews, product-design and evaluation documents, and project-management plans.

Consulting – Members of the Software Center are available to help you during any phase of your project, particularly with tools and methods for the design and development of software products.

Corporate Archiving – The Center archives Tektronix product software and assists others with procedures aimed at assuring proper archiving of all Tek software and software tools.

Tools support

Based on the evaluation of engineering needs, we formed a Tools Support Staff within the Software Center. The staff, led by Tom Milligan, has these objectives:

- Help the divisions identify, acquire and install S/W development tools
- Help engineers solve their problems in using supported tools
- Assure that error-fixes are professional and timely
- Provide an on-line catalog of available and supported tools at Tek.

A major service of the Tools Support Staff is helping project leaders develop phase-by-phase tool plans early. The staff can help you select the best tools for functionality, reliability, ease of use, level of support, and cost.

The tools staff can also coordinate in-house tool-use training. Training is done with instruction documentation or by groups within Tektronix.

UNIX documentation

Documenting UNIX presents many challenges, for example, the conversions from 4.1c to 4.2 bsd and the introduction of ECS UNIX.

To date, both ECS-related documentation and 4.2 Berkeley reprints have been published by Technical Communications (in the Scientific Computer Center). Nancy Peate, UNIX documentation project leader, now coordinates the compilation and distribution of Tektronix manual pages (local Tek mods). She also handles orders for the USENIX user's and programmer's manuals (4.2 Bsd).

UNIX documentation, textbooks, tutorials, and manuals are available from Hazel Ade in Technical Communications, 627-1771 (76-036). New publications and UNIX tutorials are announced in *CSC INTERFACE.* To receive the *INTERFACE,* call Editor Nancy Peate, 627-1763.

UNIX-user assistance

User training – The UNIX operating system is extremely powerful. It's also difficult to learn. And there are few UNIX experts at Tektronix. For these reasons, the UNIX User Assistance group provides on-site training and support to help software engineers to become proficient in UNIX quickly.

Telephone help – Users can get telephone help from the Scientific Computer Center's Applications group:

In Beaverton, call Don Hauge at 627-5245.
At Walker Road, call Byron Rendar at 629-1630.

UNIX questions can be sent via electronic mail to *tektronix!unix-help.* Walk-in customers are also welcome at all three sites.

Questions ranging from simple to complex have been answered with the help of the Small Systems Support Group (S³G). S³G is part of the Scientific Computer Center.

UNIX systems support

The UNIX Small Systems Support Group (S³G) existed before the Software Tools Program. The group's former responsibilities have been changed to support the new program.

Bob Perry, group manager, has established ways to install and support Berkeley UNIX version 4.2 on Tektronix VAX computers until the Tek standard version is ready. Upon installation of a Tek Standard UNIX, S³G will provide the Tektronix engineering community with bug fixes, extensions and other changes.

Installation of version 4.2 Berkeley UNIX is complete on most existing systems.

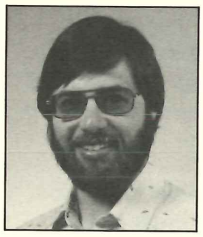
Summary

We have looked at the status and progress of the Software Tools Program and its major commitment to helping develop a Tektronix Unix, standardized and supported across product groups. Today, Tek engineers have access to training, on-line help, and documentation that just a short time ago were virtually nonexistent for UNIX. Soon more, and more powerful, software development tools will be accessible.

For More Information

For more information, call Chuck Martiny 627-6834. □

FACTORS IN DESIGNING ONLINE DOCUMENTATION



Herb Weiner is a senior software engineer in the Information Display Division. Among his responsibilities is the design of the user interfaces in IDG products. Herb serves on the Tektronix User Interface Coordination Committee. He is a member of ACM SIGDOC, SIGGRAPH, and SIGCHI, the Computer Systems Group of the Human Factors Society, and IFIP Working Group 6.3 on Human-Computer Interaction. He received his BS degree in engineering from Cornell University.

This article will appear as a chapter in the forthcoming *Handbook of Systems Documentation,* edited by Diana Patterson, to be published by VanNostrand Reinhold. This article must not distributed outside of Tektronix, Inc.

Requirements of Online Documentation

Online documentation must satisfy a more rigorous set of requirements than printed documentation. If most of these requirements can not be satisfied, the user may be better off with more traditional documentation. Some applications (e.g., installation instructions, service instructions) are not suitable for online documentation. Good online documentation requires more than just placing a manual online.

Should be as flexible as printed documentation

Online documention must provide all of the flexibilities of a printed document. The reader must be able to pick up and put aside a document at will, whether it is printed or online. Systems which require the user to terminate the task at hand (e.g., exit from an editing session) to use the documentation are not acceptable. The user must be able to skip around (flip pages) within a document, and refer to several documents simultaneously. Referencing an online document should not obscure the displayed work in progress; but if it does, it should be possible to restore the contents of the display after finishing with the online documentation.

Must be concise

Online documentation must be more concise than printed documentation because the capacity of most displays is less than a printed page, and because updating displays typically takes longer than turning a printed page. Printed documentation makes use of spacing, font styles and sizes, and graphics to separate topics, lead the reader's eye, and illustrate concepts. The ideal display for online documents would have these same capabilities. In practice, documentation must often be designed for use

on a variety of displays, some of which are barely adequate. In some situations, the display capability may be so limited (e.g., a microwave oven) that online documentation is not even feasible.

Direct access needed

Limitations in the speed with which information can be displayed necessitate a more direct means for locating information than flipping pages in a manual. These limitations are more severe when data communications speeds are the limiting factor. An online index can provide direct access; an even more direct path can be provided by keying the information to the current task. For example, pressing a HELP key could provide online descriptions of the options available to the user. Pressing the HELP key immediately after an error message could provide more information about the error and the appropriate corrective action. Users may find it convenient to place "bookmarks" at spots in the online documentation, so they can easily return to the same point later.

Users may want hard copies

Online documentation has the disadvantage that it can only be used on the system. Unlike a printed document, users can not make notes in the margins of online documents. Thus, users may want printed copies to carry away or annotate. While it is sometimes satisfactory to queue a print request to a shared printer for later delivery, users often want the hard copy right away.

Integrating online and printed documentation

Online and printed documentation should be closely integrated for information consistency and to simplify the document creation and maintenance. One approach is to write the documentation using a set of formatting macros (a general markup language). This technique allows the same information to be formatted for either online or printed documentation. If the information is concise and well organized for use as online documentation, it will probably be a good base for the printed documentation, even if the printed documents are to be more detailed.

Good online documentation needs to be closely integrated into the user interface. Only through such close integration will it be possible to key the online documentation to the task at hand.

Requirement: Serving Varied Users

It is widely recognized that printed documentation must serve a variety of users. Often, this requirement is achieved by providing multiple documents; for example, a tutorial manual, a reference manual, and a service manual. This separation is not usually obvious with online documentation, since the same information can be accessed in several ways. For example, an entire document could be presented sequentially (in small units) as part of a tutorial, while individual sections could be presented (for reference) when the user presses the HELP key. Organization of online documentation to support such varied use requires more effort in the planning stages, but can reduce the amount of writing needed.

Reference organization

Reference documentation needs to be organized to enable users to rapidly find answers to specific questions, so that they can proceed with the task at hand. The most common requirement is to determine the syntax or semantics of a specific command. Sometimes, a user may need to determine the name of a command which will perform a specific function. A good online help system will also answer questions about equipment (e.g., terminals, printers), services (e.g., rates, hours of operation), or other information (e.g., where to look, who to ask).

Tutorials

Tutorials need to provide general (overview) information as well as specific information. Prolonged usage is common, perhaps spanning several sessions, since the learning is the task at hand. Usage of a tutorial is not necessarily followed by usage of the system. However, a tutorial may involve directed use of the system. For example, the EMACS editor tutorial involves using EMACS on a special file which describes the EMACS commands, tells the user when to try them, and then explains the results. The tutorial on the Apple Lisa also involves using the system, but in an environment that provides assistance if the user makes an error.

Document authors and updaters are users too

Remember that the people who write and maintain the online documentation are users too! They need to be able to update the documentation with consistency when a system changes; they need to perform global searches and updating. If a document is divided into many small units to support direct access to the required information, it may be difficult to perform such global maintenance using the standard editor, and special maintenance tools may be needed.

User created documentation

If an online documentation system is good, users or groups of users may want to integrate the documentation for their own application into the system. Since this documentation may describe applications not available to all users, individual users should not be allowed to change the master copy of the documentation. Furthermore, they should not copy the master documentation and then update that copy, since subsequent changes to the master would not change their private copy. Users should see consistency between treatments of system-provided applications and documentation and user-provided applications and documentation. That is, the user should be able to build an environment in which his or her application programs appear to be "standard" system facilities.

Foreign language documentation

If a product is being developed for international markets, foreign-language documentation can be important. For this reason, it is a good idea to avoid English-language dependencies in the design of online documentation. For example, compact hash tables or alphabetical organizations optimized for English keywords may not operate well (or at all) after translation to a foreign language. Such techniques should be avoided anyway, since they make it more difficult to extend or modify the system, even in English.

Computer Center Rates		
Period	Item	Rate
8:00 AM to 5:00 PM	CPU Time	100.00/Hour
	Connect Time	25.00/Hour
5:00 PM to 8:00 AM	CPU Time	50.00/Hour
	Connect Time	10.00/Hour

Figure 1. A table constructed using ASCII characters.

Online documentation not always appropriate

Online documentation is not appropriate for every product. It requires storage, whether disk-based or memory-based. If online documentation can be shared by users, each of whom would otherwise need their own copy of the printed documentation (as in a host-based, multi-user system), online documentation can be less expensive than printed documentation. Conversely, on a single-user system, online documentation might be more expensive than printed documentation. Although this higher cost may be justified by increased productivity the cost-benefit tradeoff should be evaluated for each situation.

Forms of Online Documentation

The appropriate form of online documentation depends upon display and hardware capabilities, the characteristics of the user, and the information to be communicated.

Text

Textual documentation is the most common form of online documentation. It is the easiest to create and maintain. Furthermore, it requires the least operating system support while supporting the widest variety of terminals. Finally, text can generally be displayed more rapidly than other forms of online documentation.

Graphics

Graphic documentation more effectively communicates certain information than does pure text. Graphic communication does not necessarily mean pictures! As with print, proper spacial placement of text can *graphically* improve effectiveness. Lists (one item per line) are the most effective way to present alternatives, even though lists leave a lot of blank space on a display which already has a very limited capacity. If capacity is a problem, multiple columns improves display utilization.

Some information is well suited for presentation using a table. The ASCII characters "*" (42), "-" (45) and "|" (124) can be used to construct elaborate tables, even on displays which do not support graphics or have a ruling-character font. Figure 1 shows an example of such a table.

Documentation employing graphs is less common than documentation containing tables. Nevertheless, such graphics can be presented, in a crude fashion, even on non-graphic displays; such a technique has long been used to create graphs on character printers. Figure 2 shows such a graph.

Sophisticated graphics

Displays with more graphics capabilities can be used for line illustrations; more sophisticated displays may support filled areas, shading, and/or color. The XEROX Star allows simple animation. One such animation documents the paper path through a copier for service training. The PLATO system allows 35 MM slides to be projected from behind the display and combined with computer-generated graphics. Another possibility is to combine camera video with computer graphics.

Audio

A less common form of online documentation is audio. Audio documentation should be considered where the eyes are not available to examine documentation (e.g., while operating machinery) or where display is limited (e.g., a microwave oven). Audio should not be used where it would distract others or where the user might not accept it (psychologically). In preparing audio documentation, voice (e.g., male and female, authoritative or friendly) is an additional factor to consider.

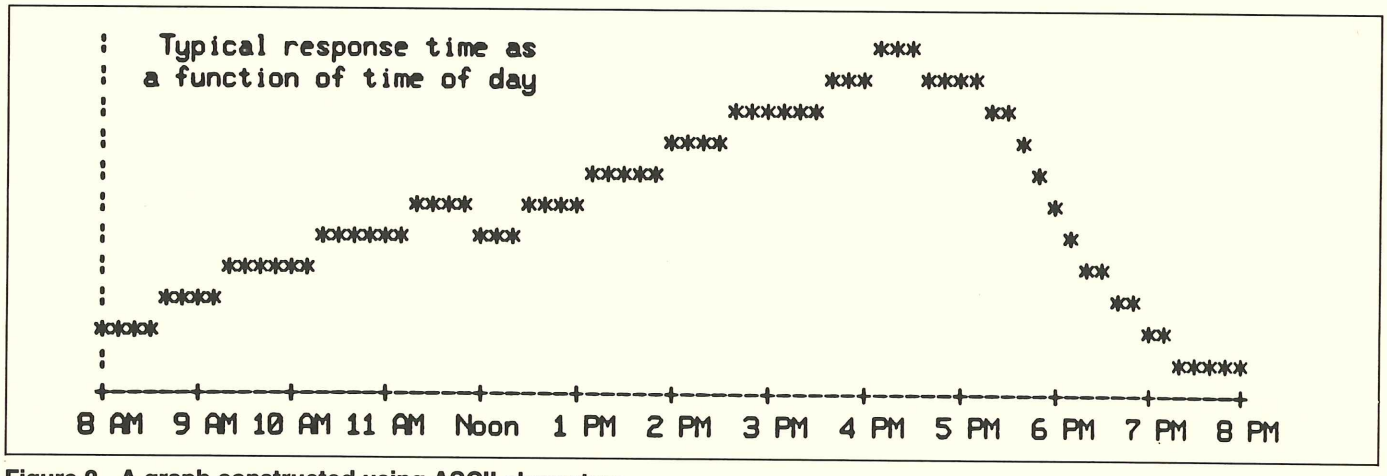


Figure 2. A graph constructed using ASCII characters.

Loosely Coupled Online Documentation

It is convenient to classify online documentation into three categories based upon the relationship between the information contained and the system on which the documentation is implemented. The first category encompasses documentation which is loosely coupled to the system on which it is implemented. Such documentation can provide information about the system on which it is implemented, or it could provide information about a different system or even computer-based tutorials unrelated to computers. (The other two categories, input assistance and error messages are closely coupled to the system on which they are implemented. These categories will be discussed in the next section.)

Sequential organization

Loosely coupled documentation can be organized sequentially, hierarchically, or randomly. The distinction depends upon the nature of the pointers, or cross references, which the system can follow to locate the desired information. Such pointers are analogous to a "see page 23" or "see section 5.2" in printed documents. Sequentially organized documentation does not contain such pointers; consequently, only complete units of the documentation can be displayed, starting at the beginning. Thus, to obtain any information about a particular command, it would probably be necessary to display all documentation for that command.

The UNIX system includes a sequentially organized online documentation system. The MAN command displays all documentation for the specified command(s). The WHATIS command is similar to the MAN command, but it displays only the summary line describing the basic purpose of the command. The APROPOS command searches the summary lines of each command and displays the summary lines which contain the specified keyword. For example, APROPOS EDITOR displays the summary line for all commands containing the word EDITOR in the summary. The WHEREIS command can be used to locate the source, binary, and/or manual file(s) for the specified command.

Hierarchical organization

Hierarchically organized documentation can be accessed by major headings, as in a traditional outline; for example: Purpose, Syntax, Files Referenced, Options, Algorithms Used, Errors Generated, and Related Commands. Each major topic could be further subdivided; for example, the documentation for a compiler would probably contain a separate section describing the source file, the listing file, and the object file. If the user requested information on FILES, the system could then provide an overview of the three files used and, if requested, details on a specific file. A user should not be required to know or specify the fully qualified path to the desired information when using a hierarchically organized online system. Thus, HELP PASCAL LISTING should be sufficient to obtain information on the listing file; HELP PASCAL FILES LISTING should be acceptable but not required. Context plays an important role. Thus, after requesting HELP PASCAL FILES, HELP LISTING should be sufficient to obtain information on the PASCAL LISTING file.

Hierarchically organized documentation does not necessarily have to be implemented using pointers. For example, the Cornell University HELP system is hierarchically organized from the users' perspective. However, it is implemented using sequential files which contain the hierarchical information on special control lines; the desired information is located by searching through the sequential files rather than following pointers.

Random organization

The pointers in randomly organized documentation are not restricted to the hierarchical organization described above; they can point to information at the same, higher, or lower level in the same or another document. Thus, information on the PASCAL LISTING file described above may reference (point to) information on the PASCAL CROSSREFERENCE option, which affects the amount of information written to the listing file. Similarly, information on the PASCAL CROSSREFERENCE option would reference the information on the PASCAL LISTING file. On the display, such references might be indicated by an asterisk (analogous to a footnote). Ideally, it should be possible for the user to follow such a reference simply by pointing to the asterisk. A button would be provided to allow (but not require!) the user to return to the previous context after following such a reference.

The Xanadu system, first described by Ted Nelson in 1974, is the first system to support randomly organized documentation, or hypertext, as Ted Nelson calls it. Xanadu even allows pointers between alternative and/or previous versions, and allows alternative and/or previous versions to be compared to locate differences at the word level. These pointers can also be used to construct new documents which logically encompass portions of existing documents without actually copying the existing information.

Closely Coupled Online Documentation

The other two categories of online documentation involve assistance in input entry and error messages; these messages provide assistance after an error has occurred. Both of these categories of online documentation must be closely coupled to the system on which they are implemented. Unless the design of such online documentation begins early in the design phase, the results will almost certainly be less than satisfactory.

Command entry

Input assistance can assist in command entry. One approach is to use traditional keyboard entry of commands, and to provide special keys that the user can press for assistance. Digital Equipment Corporation used this approach in designing the TOPS-20 operating system. During command entry, the user can enter a question mark and the system will immediately respond by telling the user what is expected. If an option is expected, the system will respond with a list of the valid options. Unfortunately, the responses are sometimes vague. For example, if a question mark is entered to determine the parameters of the COPY command, the system will respond that a filename is expected, but it will not specify whether the expected filename is the source or the destination.

If the escape key is pressed during command entry, the system will respond by completing the current field if the characters already entered are sufficient to uniquely determine the user's intent; otherwise, the system will ring the bell on the terminal. For example, if the user types COP and presses the escape key, the system will type the Y. The escape key also works when entering filenames or options. Unfortunately, if the entry is ambiguous, the system will not list the possibilities.

If the user presses the backspace key immediately after receiving an error message resulting from an error in entering the command, the system will retype the command up to, but not including the error. Unfortunately, if the error is near the beginning of a long command, the user must retype most of the command.

Menus

An alternative to traditional command entry is command entry from menus. Because menus change the task of command entry from one of recall to one of recognition, menus are a form of documentation. For example, a user unfamiliar with a system may not recall how to produce the final copy of the documentation after completing some editing. However, that user would probably recognize PRINT as the correct command if he saw it on a menu.

System designers should consider the fact that people have different learning styles. Some people are more successful with tasks described in words, while others respond more favorably to spacial formats. An advantage of menus over traditional command entry (words) is that both spacial and verbal cues are presented simultaneously. In other words, a spacially oriented user will always find a specific command at a specific position in the menu.

Usually, menus are displayed on the screen. If special graphic input hardware is available, the user can pick a menu item by pointing to it. Otherwise, the user must enter the selection using the keyboard, usually by typing the item number of the desired selection. If the quantity of possible commands is large, hierarchical menus are often used. The upper-level menu might allow users to select editing commands, printing commands, backup utilities, etc. If the user selects printing commands, the next-level menu might allow the user to select the paper, specify the number of copies, specify the name of the file to be printed, etc.

Menus and the time to invoke a command

The time required to invoke the command is a function of three factors: (1) the time required to display the menu, (2) the time required for the user to search through the menu and make a selection, and (3) the time required for the user to indicate the selection to the system.

In contrast, the time required to invoke a command using traditional command entry is a function of the user's think time and the time required to key in the command. For beginners, think time is the dominant factor, and menus can reduce this think time. For experts, however, display time is the dominant factor, and menus can hinder productivity. This is especially true when an experienced user must pass through several menus in a system having a hierarchical menu organization in order to get to the desired command. (Menus are like having help information displayed all the time, even when it is unnecessary.)

Tablet menus

Graphic input tablets can provide an alternative to screen menus. A tablet menu is printed on paper, mylar, etc. which is fastened to the tablet surface. Entries are selected by pointing to the menu item on the tablet, without using the display or keyboard. Since a printed menu is used, the menu can group related items within boxes, and can employ graphic information or be color coded. *The most significant advantage of tablet menus is that display-time delay is eliminated.* However, since the position of the menu on the tablet is critical, switching between several menus is not practical. Also, tablet menus can not be used with other graphic input devices such as a mouse. In situations where a tablet is already being used for graphic input, tablet menus are suitable for experts as well as beginners, since they can reduce the need to switch frequently between keyboard entry and tablet entry.

Input assistance

Assistance can be provided for entering data as well as for entering commands. Where data is to be entered in a fixed order, prompting can be used, but form-fillout is more flexible. However form-fillout requires special support in the terminal. The system constructs a form on the display, complete with spaces to be filled in by the operator. The operator can fill in the spaces in any convenient order, and can even leave some spaces blank, if the application allows.

Error messages, the user's feelings

Error messages provide online assistance after an error is detected. The error-message designer must understand the user's point of view in this situation: The message receiver's work flow has just been interrupted, and he or she may have to correct something to proceed with the task at hand. This interruption may produce an emotional state (frustration, stress, surprise, anger, and/or fear) which makes successful communication more difficult.

Error message guidelines

Error messages should provide the user with sufficient information, in the user's terms, to identify and correct the problem, unless the information is not available or being intentionally withheld (like a password). For example, the message "Insufficient space to store file on disk" does not provide information sufficient to correct the problem. A better message might be "Insufficient space to store file on disk. 103 additional disk blocks required." All messages should be as semantically accurate as possible. For example, "File not found" may imply to an inexperienced user that the file does exist, but has somehow been lost. A better message would be precise: "File does not exist."

The error messages should be understandable to the user without referring to additional documentation. For example, the error message "DISK21 Insufficient space 103,534" cannot be interpreted without additional documentation. The message could mean "Insufficient space to store file on disk. 103 blocks available of 534 required [431 more needed]" or "Insufficient space to store file on disk. 103 additional blocks required of 534 total" or even "Insufficient space to store file on disk. 103 blocks available; 534 additional blocks required [637 total]."

If the system is designed for users with widely differing experience, the error message should be brief, and indicate that more information is available. Present the most frequently required information first. There are several reasons for choosing this strategy. First, novices are generally willing to ask for additional information, provided that information is available; second, many expert users object strongly to systems which treat them like beginners. Furthermore, even novices need to see a particular message only a few times before they understand that message; however, they are likely to see the message many more times after they become experienced – and as an "expert," react negatively to over-information.

For example, the message regarding disk space (above) might be changed to read "Insufficient space to store file on disk. 103 additional disk blocks required. Press HELP for more information." If the user presses HELP, the system might respond "Either delete some files or insert a different disk." It is this additional message which tells the user how to correct the problem. Ideally, the error-message system should both contain pointers to other portions of the online documentation and consider the context in which the user presses HELP. Thus, if the user presses HELP again, the system might list only those commands relevant to the current situation; for example: obtain a list of the files which can be deleted, examine a particular file to determine whether or not to delete it, and delete specific files.

Finally, it is important for messages to be consistent in how they indicate the severity of the problem. For example, error messages (which require explicit action) should be clearly distinguished from warning messages (which may often be ignored).

For More Information

For more information, call Herb Weiner 685-3586. ☐

LIBRARY VIDEO TAPES COVER AI, INGRES, AND DIGITAL SIGNAL PROCESSING

The Library in building 50 has added three video-taped courses to its collection. These 3/4 inch tapes may be viewed in the Library or they may be borrowed.

Twenty-three tapes make up a course on artificial intelligence given by Patrick Winston, MIT's AI authority. The text for this course is *Artificial Intelligence,* by Patrick Winston, published by Addison Wesley (about $23.00). The text is not essential for the first part of the course, which covers fundamentals. Because the Library does not circulate the text, you will have to buy it to complete the course.

Twenty-two tapes teach an MIT course titled "Digital Signal Processing." The Course is taught by Alan Openheim, whose book *Digital Signal Processing* (Prentice-Hall, $33.00) is used as the text. This book too, must be purchased.

The third new taped course is on INGRES, a relational data management system. Twelve tapes make up the course.

For more information, call the Corporate Library, 627-5388. ☐

Technology Report
MAILING LIST COUPON

☐ ADD
☐ REMOVE

Not available to field offices or outside the U.S.

MAIL COUPON TO 53-077

Name:_____ D.S.: _____

Payroll Code: _____

(Required for the mailing list)

For change of delivery station, use a directory change form.

TEKTRONIX STANDARDS NEWSLETTER

Technology Report will no longer publish routine information about new standards and standards activity. This information is now available in the *Tektronix Standards Newsletter.* To receive the newsletter, contact Bonnie Kooken, 627-1799 (78-529). □

COMPANY CONFIDENTIAL
NOT AVAILABLE TO FIELD OFFICES

TECHNOLOGY REPORT

RICHARD E CORNWELL

mail label goes here

19-285

DO NOT FORWARD

Tektronix, Inc. is an equal opportunity employer.