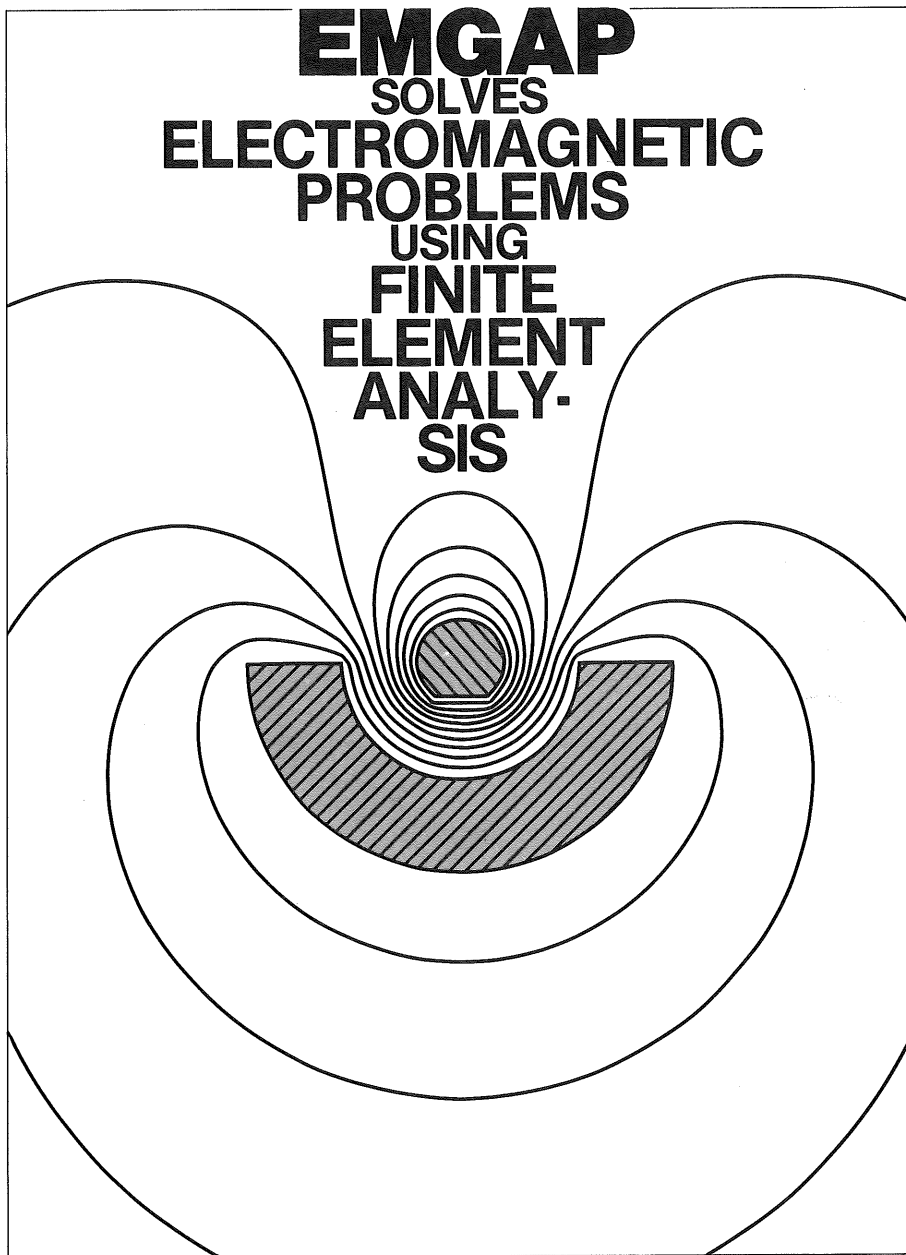


TECHNOLOGY report

COMPANY CONFIDENTIAL

EMGAP
SOLVES
ELECTROMAGNETIC
PROBLEMS
USING
FINITE
ELEMENT
ANALY-
SIS



CONTENTS

EMGAP Solves Electromagnetic Problems Using Finite Element Analysis	3
High Level Simulation of Digital System Using N.mPc	11
Engineering Activities Council Membership Changes	18
Modula-2 As a Software Engineering Tool	19
Book by Tek Author Deals with Standards	22
Ergonomic Barriers Are Real	23
User Interface Aspects of a Desktop CAD System	25

Volume 5, No. 2, May 1983. Managing editor: Art Andersen, ext. MR-8934, d.s. 53-077. Cover: Joe Yoder; Graphic illustrator: Jackie Miner. Composition editor: Jean Bunker. Published for the benefit of the Tektronix engineering and scientific community.

Copyright © 1983, Tektronix, Inc. All rights reserved.

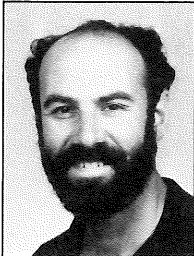
Why TR?

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).

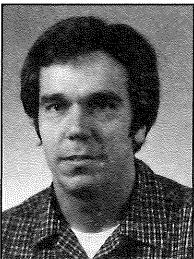
Contributing to TR

Do you have an article or paper to contribute or an announcement to make? Contact the editor on ext. MR-8934 or write to d.s. 53-077.

EMGAP SOLVES ELECTROMAGNETIC PROBLEMS USING FINITE ELEMENT ANALYSIS



Jeffrey Beren supervises electromagnetic modeling in High Frequency Component Development, part of the Solid State Group. Jeff joined Tek in 1977. He received his BSEE from Washington University (St. Louis, Missouri); his MSEE from the University of Arizona (Tucson); and his PhD in electrical engineering from the University of Arizona.



Bob Kaires is an electrical engineer doing electromagnetic modeling within High Frequency Component Development, part of the Solid State Group. He joined Tek full time in 1979 after summer interning in 1978. Bob received his BS in physics from Rutgers College (New Jersey) and his MS in physics and MS in electrical engineering from Michigan State University.

Introduction

The Electromagnetic General Analysis Package (EMGAP) is an evolving collection of software programs that allows the user to solve electromagnetic problems in 1, 2, and 3 dimensions.

The package is divided into three major parts (see figures 1 and 2). The input and output parts are handled by pre- and post-processor programs, which reside on a VAX computer optimized for interactive graphics. The numerical analysis program resides on a Cyber computer optimized for numerical analysis. In this configuration, additional preprocessors, postprocessors

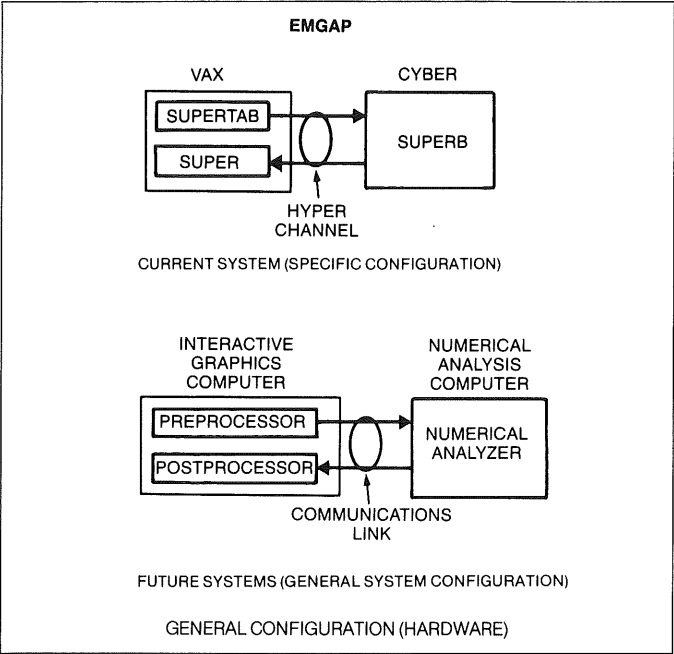


Figure 2. and numerical analyzers can be added to EMGAP as the need arises.

EMGAP Today and Tomorrow

The current version of EMGAP is a collection of Structural Dynamics Research Corporation (SDRC) software designed for

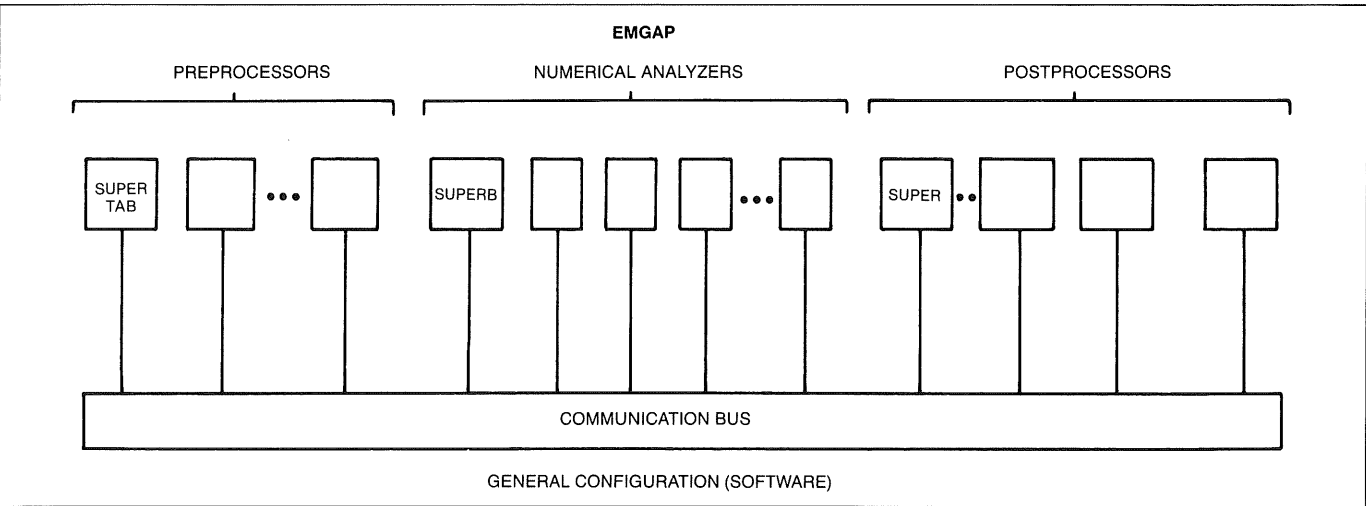


Figure 1.

mechanical engineering finite element analysis.^[1-7] (This software is marketed by General Electric, Computer-Aided Engineering International.) Because the finite element technique is a very general numerical technique (see The Finite Element Method), nothing inherently restricts the technique to mechanical engineering. Many of the mathematical equations that describe mechanical engineering problems also describe electrical engineering problems.

In the current version of EMGAP, we use the SDRC software that solves the mechanical engineering heat problem (Laplace's equation). For electromagnetics, the heat problem can be interpreted as the electrostatics problem. Thus, we can make correspondence between isotherms and equipotentials, heat flux and electric flux, heat source and electric charge.

In EMGAP, the SDRC preprocessor program SUPERTAB handles interactive graphic input. Interactive graphics output is handled by SDRC postprocessor program SUPER, which plots the problem geometry, equipotentials (isotherms), and lines of constant-D field (heat flux) magnitude.

The numerical analysis is done with SDRC analysis program SUPERB. SUPERB solves Laplace's equation (electrostatic and magnetostatics problems in source-free regions), and we have modified it to calculate electrostatic capacitance (or magnetostatic reluctance). SUPERTAB and SUPER temporarily reside on the HCAD VAX with the VMS operating system, and SUPERB resides on the CYBER A machine (figure 2).

In future versions of EMGAP, Poisson's equation (electrostatics and magnetostatics problems in regions containing sources) and Helmholtz's equation (electrodynamics) will be available. In addition, the organization of the package is versatile enough to let us add software other than mechanical engineering software. Thus specific programs written directly for electrostatic application^[8] could be added to the package. The software added could then employ SUPERTAB to set up the problem geometry. However, future development depends on how users respond to the current version of EMGAP and what future capabilities users want.

Applications and Capabilities

Some of the 2-D capability of EMGAP now includes shielded transmission lines of arbitrary cross-section, magnetic relays, gallium arsenide (GaAs) bridges and air lines, and high voltage structures.

A shielded microstrip problem is an example of an EMGAP solution of a shielded transmission line of arbitrary cross-section. Figure 3 shows the element arrangement of half of a shielded microstrip transmission line. Only half of the geometry need be entered because of the symmetry plane perpendicular to, and bisecting, the microstrip. Figures 4 and 5 show the equipotentials for substrate dielectric constants (ϵ_r , equal to 1 and 10, respectively).

For the substrate with $\epsilon_r = 10$, a closed-form solution, accurate to 1%, gives a capacitance of 178.074 pF.^[9] Using 60 cubic elements, EMGAP gave a capacitance value of 181.507 pF, a difference of 1.9%. The problem was solved again using 150 cubic elements, reducing the difference to less than 1%.

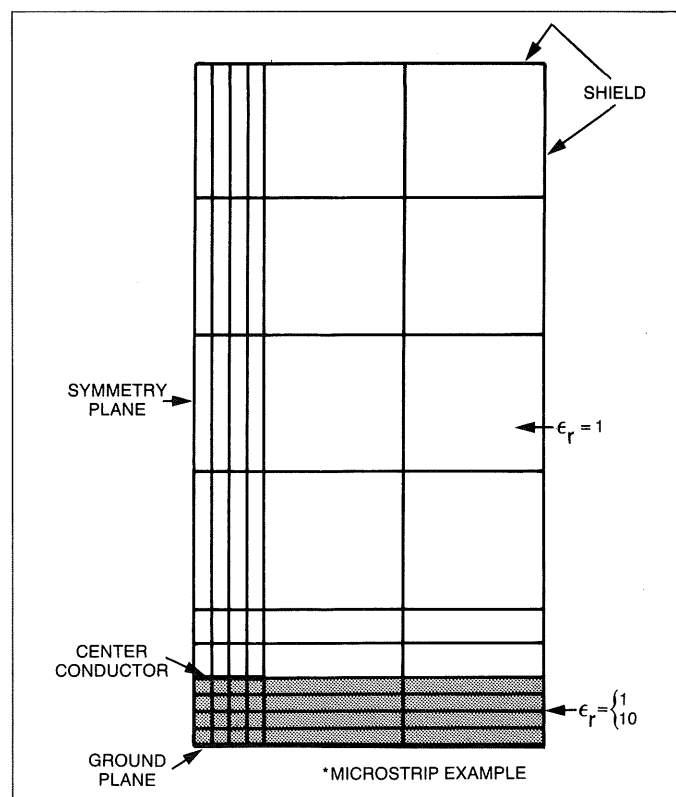


Figure 3.

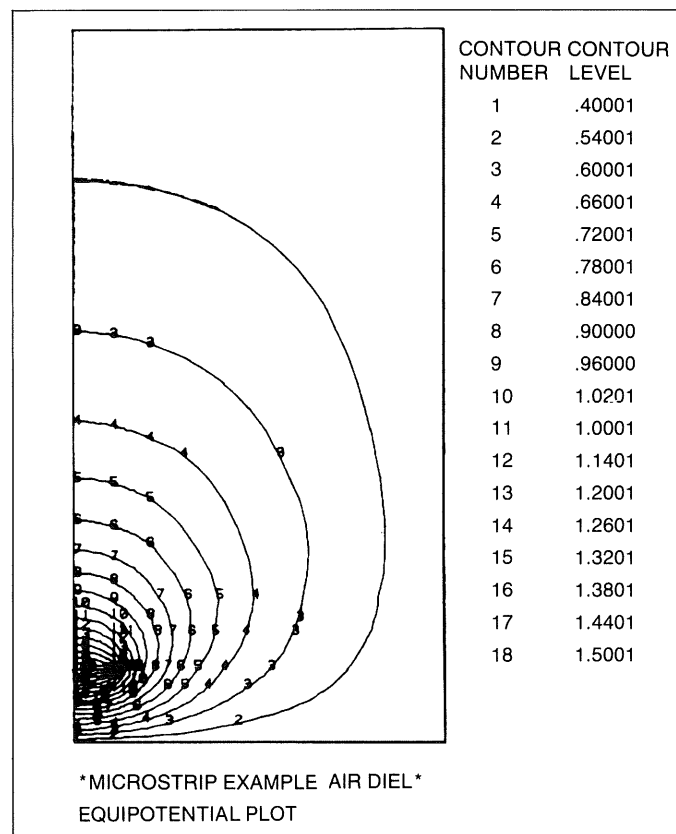


Figure 4.

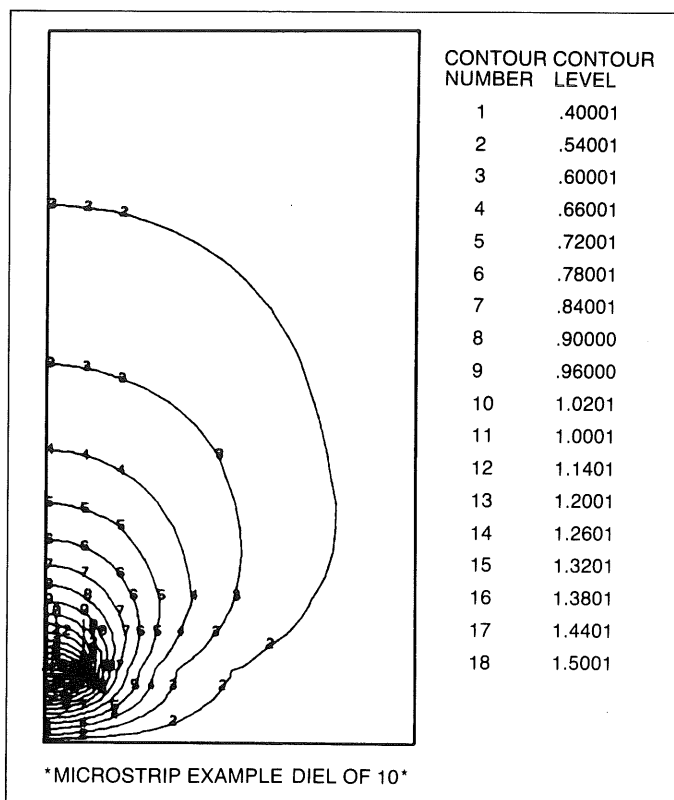


Figure 5.

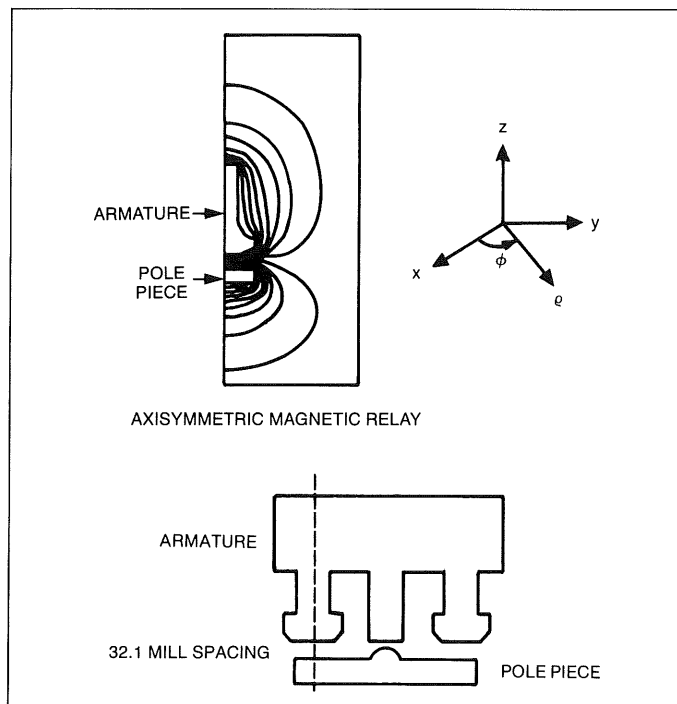


Figure 6.

For the case of substrate dielectric $\epsilon_r = 1$, EMGAP gives a value of capacitance of 27.491 pF. Using this value and the one above for $\epsilon_r = 10$, we calculated the characteristic impedance for the shielded microstrip as 47.67 ohms.

Another EMGAP application involves an axially symmetric magnetic relay. Figure 6 shows a radial slice of a 3-dimensional object used to model a magnetic relay. The solid can be generated by rotating the slice about the Z axis. SUPERB handles this as a 2-dimensional problem because of the symmetry.

Figures 6 and 7 show lines of constant magnetic potential. Figure 7 is a magnification of an interesting area that also shows element density. From repeated solutions of this problem for different armature/pole piece separations, we produced a curve of reluctance vs. spacing (figure 8).

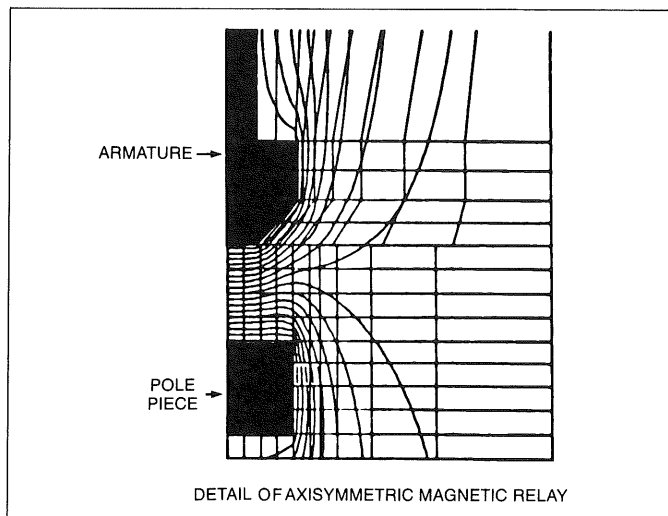


Figure 7.

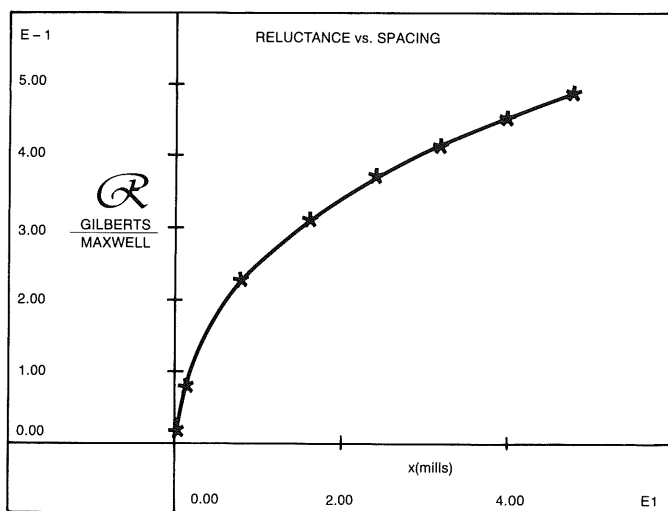


Figure 8.

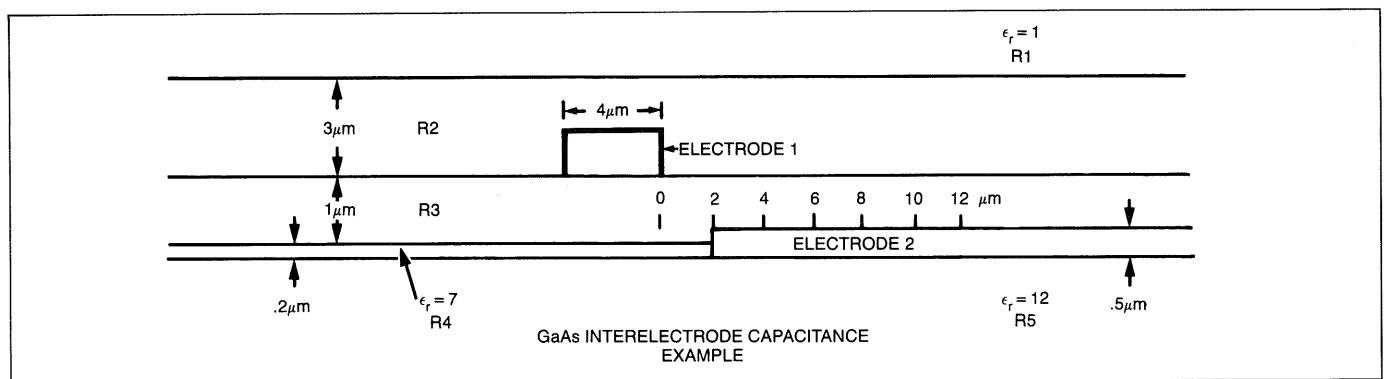


Figure 9.

The GaAs bridges and air lines example demonstrates the ability of the EMGAP program to handle a problem with more than one dielectric region. Figure 9 shows five different dielectric regions. For one such interelectrode configuration — the air bridge — three of the five regions (R1, R2, and R3) have air dielectric constants. Figure 10 shows the equipotentials and finite element density for this analysis. If R3 has a dielectric constant other than air, the same model (figure 9) represents a bridge conductor resting on a dielectric support. For various spacings of electrode 1 to electrode 2 and for different dielectrics, it was possible to develop design curves for the interelectrode capacitance (figure 11). This information can be used to help design GaAs circuits.

EMGAP also solves high-voltage analysis problems such as a wire over a ground plane (in a box) (figure 12). We can set up the model to minimize the effects of the sides and top of the box (figure 13). Windowing in on figure 13, we can see the details of the equipotential lines for a line above a ground plane (figure 14). SUPERB calculates field strength, field energy, and potentials. For more complex problems, we can study the effect the model's geometry has on these parameters.

A shielded square capacitor problem demonstrates the 3-D capability of the program (figure 15). For this example, we considered the dielectric region to be air. The element density and equipotential contours are presented at various cross-sections: (1) in the plane of the square plate (figure 16), (2) in a plane parallel to the preceding case but one unit above the plane of the square plate (figure 17), and (3) in a perpendicular plane bisecting the square plate (figure 18). Note the similarity of the equipotentials for the bisecting plane (figure 18) and the equipotentials for the shielded microstrip problem (figure 4).

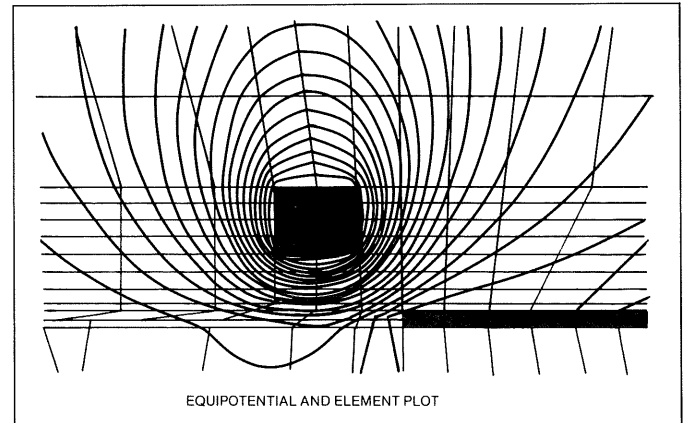


Figure 10.

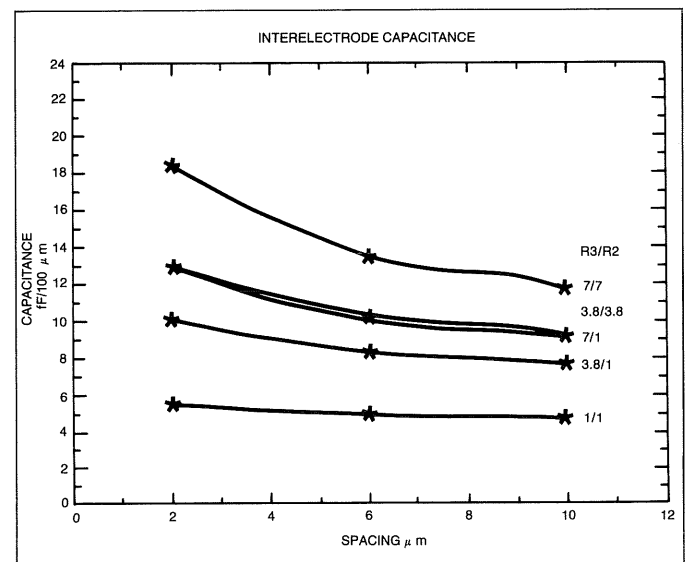


Figure 11.

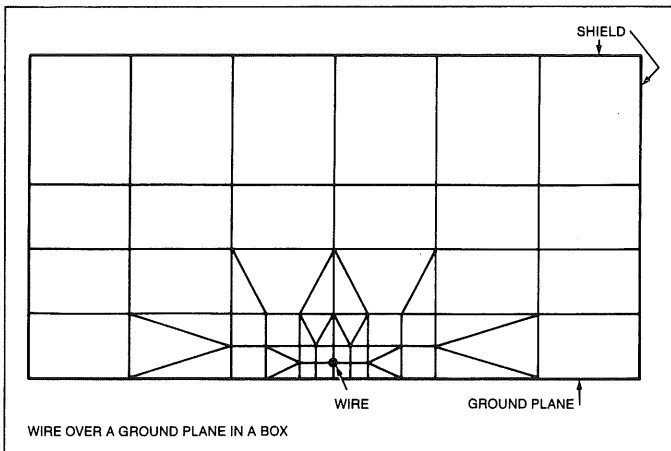


Figure 12.

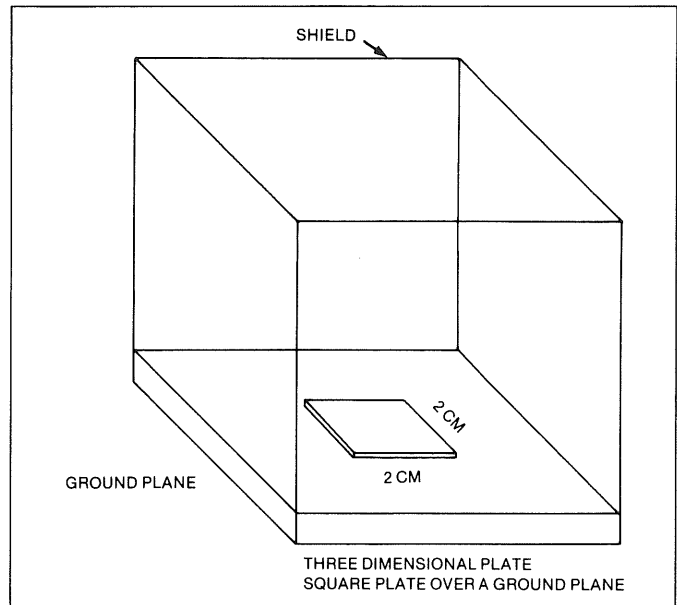


Figure 15.

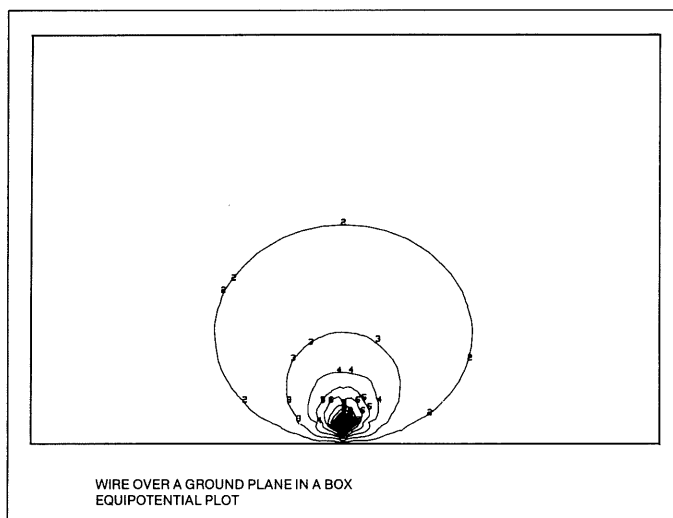


Figure 13.

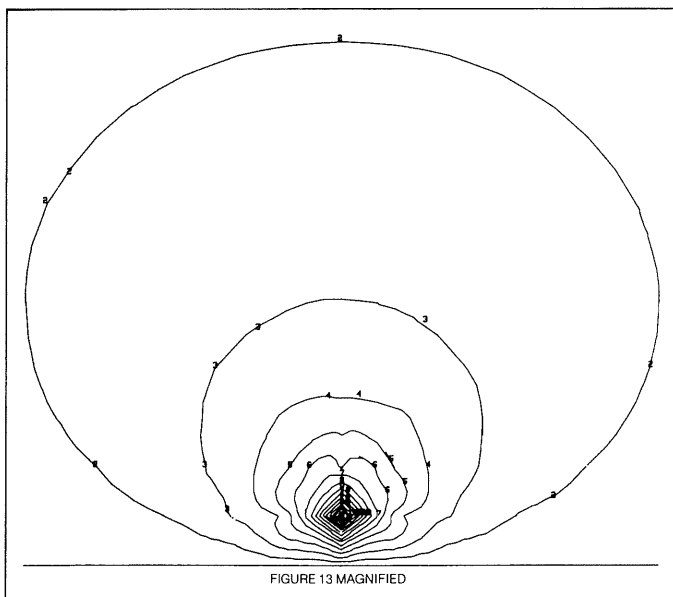


Figure 14.

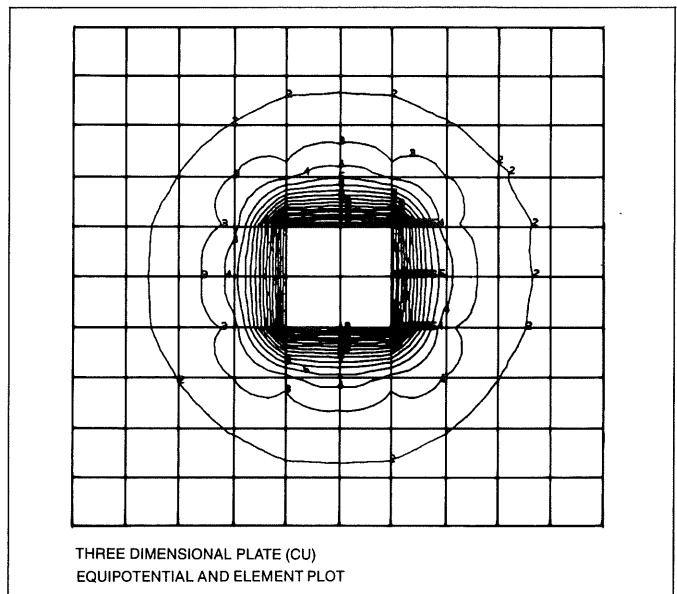


Figure 16.

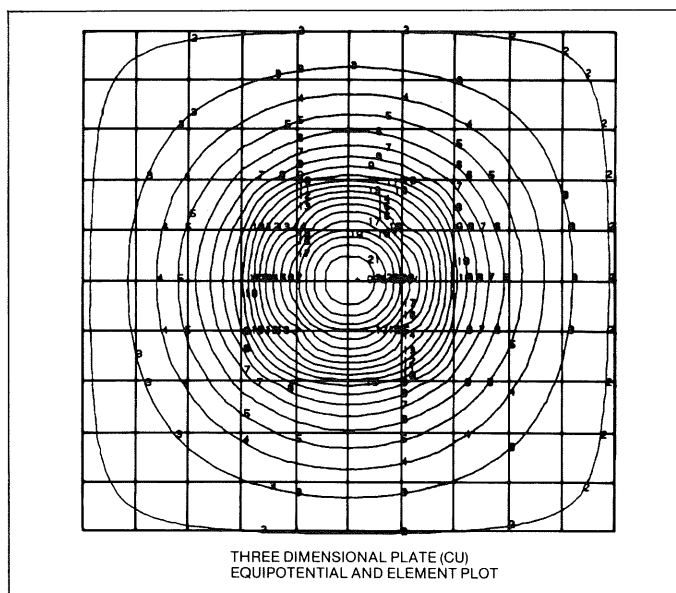


Figure 17.

The square capacitor example has wider application. If the square plate is replaced by several rectangular strips, this example could represent a shielded meander line in the quasi-static approximation and would apply to CRT deflector analysis. If the rectangular plate is replaced by two rectangular plates of different widths, the example could represent the quasi-static approximation to a step in a microstrip line. An equivalent circuit could then be derived and the resulting equivalent circuit could be used in GLUMP or SUPER COMPACT. This, in turn, would lead to more accurate high-frequency analysis.

How to Operate EMGAP

To use EMGAP, users need a HCAD VAX user number and a CYBER A user number. A beginner's user manual is available from Carolyn Schloetel, ext. B-1762. EMGAP temporarily resides on HCAD VAX and may be moved in the future. However, such a move will not affect usage.

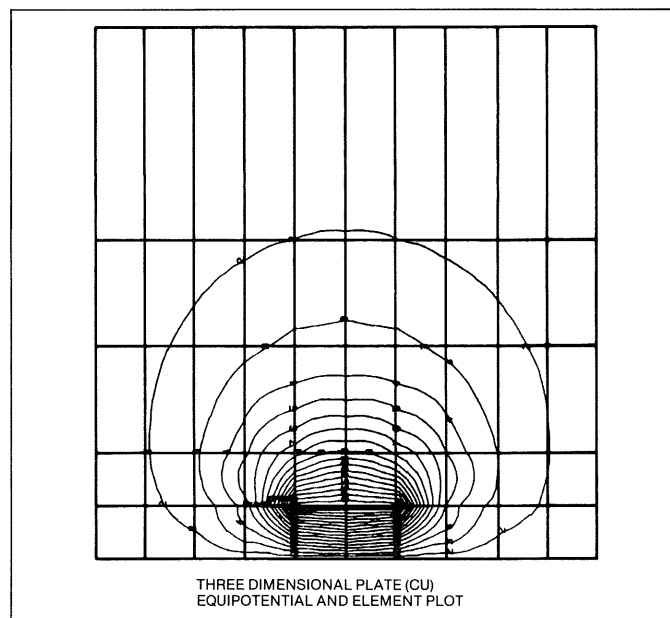


Figure 18.

For More Information

For more information, call Jeff Beren ext. B-3128 or Bob Kaires ext. B-3150. □

Acknowledgement

We would like to thank the CAE Development Group (Jack Hurt, Barry Ratih, Matt Reddy, and George Chang) for their continued support during the development of EMGAP. Special thanks to Barry Ratih whose support and patience were critical to the success of the project.

THE FINITE ELEMENT METHOD

The finite element method (FEM) is a numerical technique for solving differential or integral equations. The distinguishing feature of this technique is that a 3-dimensional region of space is divided into subregions (or elements). This subdivision facilitates the numerical treatment.

Different finite element approximations arise from mathematical formulations such as variational principles, weighted integral expressions, Lagrangian multipliers. The finite element method discussed next will be of the variational type and will be used to solve Laplace's equation.

Laplace's equation, $\nabla^2\phi=0$, together with boundary conditions uniquely specifies the electric potential ϕ in a region of space. Another way to find electric potential is by minimizing the energy integral:

$$W(\phi) = \frac{1}{2} \int_V |\nabla\phi|^2 dv$$

This is called the minimum energy principle. The potential ϕ , which minimizes the energy integral, also satisfies Laplace's equation with boundary conditions. In fact, these two methods of finding ϕ can be shown to be mathematically equivalent.

The energy integral can be minimized using the Rayleigh-Ritz method in which an approximate potential function:

$$\phi = \phi(\alpha_1, \alpha_2, \alpha_3 \dots)$$

is specified.

The parameters, α_n , are chosen such that

$$\frac{\partial W}{\partial \alpha_n} = 0 \text{ for all } n.$$

This results in n simultaneous equations for n unknowns. Using the boundary conditions along with the n simultaneous equations leads to the determination of the α_n 's.

The finite element method discussed here is an extension of the Rayleigh-Ritz technique. The region of interest is divided into elements and Rayleigh-Ritz technique is applied over each element. The potential is then approximated over each element by a function, in this case a polynomial expansion:

$$\phi^{(e)} = \phi^{(e)}(\alpha_1^{(e)}, \alpha_2^{(e)}, \dots)$$

A number of spacial coordinates, called nodes, are identified in each element. The polynomial expansions are formulated in terms of the unknown potentials at these nodes. The "nodal potentials" which minimize the energy expression are then found.

Two features that distinguish the finite element method from other methods are:

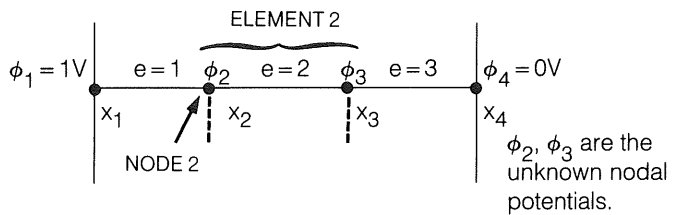
1. In finite elements, once the nodal potentials are known, the potential field throughout the element is described by the approximating polynomial. In contrast with this, the *finite difference method* — a well-known numerical technique — formulates the solution at discrete points in space.
2. The minimum energy principal is a stationary variational formula. This means that first-order small errors in ϕ lead to second-order small errors in energy. We can therefore expect that quantities derived from energy (capacitance for example) are more accurate than the electric potential solution might indicate.

The following one-dimensional example illustrates the principles of the finite element method:

Consider the one dimensional case where the potential only varies in one direction (an infinite parallel-plate capacitor, for example). To within a constant, the electrostatic energy is equal to:

$$W(\phi) = \frac{1}{2} \int_V |\nabla\phi|^2 dv$$

Further, let's divide the region into three linear elements:



$$W = \frac{1}{2} \int_V |\nabla\phi|^2 dv = \Delta y \Delta z \int_{x_1}^{x_4} \left| \frac{\partial\phi}{\partial x} \right|^2 dx$$

$$W = \sum_{e=1}^3 W_e \quad W_e \text{ is the elemental energy}$$

$$W_e = \Delta y \Delta z \int_{x_e}^{x_{e+1}} \left| \frac{\partial\phi^{(e)}}{\partial x} \right|^2 dx$$

The approximating polynomial over element e is:

$$\phi^{(e)}(x) = ax + b.$$

The nodal potentials are:

$$\begin{aligned} \phi^{(e)}(x_e) &= \phi_e \\ \phi^{(e)}(x_{e+1}) &= \phi_{e+1} \end{aligned}$$

We can reformulate $\phi^{(e)}(x)$ in terms of these nodal potentials as:

$$\phi^{(e)}(x) = \phi_e \left[1 - \frac{x_e - x}{x_e - x_{e+1}} \right] + \phi_{e+1} \left[\frac{x_e - x}{x_e - x_{e+1}} \right]$$

The integrand of the elemental energy integral can now be found:

$$\left| \frac{\partial \phi^{(e)}}{\partial x} \right|^2 = \left| \frac{\phi_e - \phi_{e+1}}{x_e - x_{e+1}} \right|^2 = \frac{\phi_{e+1}^2 - 2\phi_e \phi_{e+1} + \phi_e^2}{L^2}$$

Here we've assumed that all elements are equal in length to L , therefore:

$$W_e = \Delta y \Delta z \frac{\phi_{e+1}^2 - 2\phi_e \phi_{e+1} + \phi_e^2}{L}$$

$$\text{and } W = \sum_{e=1}^3 W_e = \frac{\Delta y \Delta z}{L} \sum_{e=1}^3 \phi_{e+1}^2 - 2\phi_e \phi_{e+1} + \phi_e^2$$

$$\text{for minimization } \frac{\partial W}{\partial \phi_2} = 0, \frac{\partial W}{\partial \phi_3} = 0$$

These two constraints together with the two boundary conditions $\phi_1 = 1$ and $\phi_4 = 0$ leads to the matrix equation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\text{the solution is: } \phi_2 = 2/3v \\ \phi_3 = 1/3v$$

Substituting back into the expression for $\phi^{(e)}(x)$ for $e = 1, 2, 3$; $\phi(x)$ is now known element-by-element over the entire region.

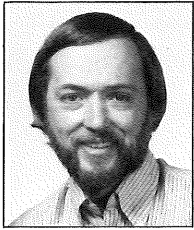
$\phi(x)$ will be continuous over the region and $\frac{\partial \phi(x)}{\partial x}$

will be piecewise continuous, there being a discontinuity allowed at element boundaries.

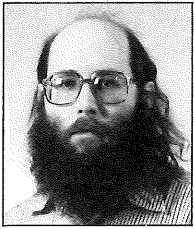
References

1. Barry Ratih, Kurt Krueger, "Finite Element Analysis Solves Mechanical Design Problems," *Engineering News*, January 1979.
2. Farid Durrani, "Finite-Element Analysis: An Available and Useful Mechanical Engineering Technique," *Technology Report*, December 1982.
3. Peter P. Silverster, Ronald L. Ferrari, *Finite Elements for Electrical Engineers*, Cambridge University Press (forthcoming latter half of 1983).
4. Isaac Fried, *Numerical Solution of Differential Equations*, Academic Press, New York 1979.
5. Gilbert Strang, George J. Fix, *An Analysis of the Finite Element Method*, Prentice-Hall, Inc., Englewood Cliffs, NJ 1973.
6. Larry J. Segerlind, *Applied Finite Element Analysis*, Wiley, New York 1976.
7. O.C. Zienkiewicz, *The Finite Element Method in Engineering Science McGraw-Hill*, London 1971.
8. John B. Rettig, "For the Designer: A General Purpose Capacitance Calculator Run from HCAD," *Technology Report*, May 1982.
9. I.J. Bahl, "Use Exact Methods for Microstrip Design," *Microwaves*, December 1978.

HIGH LEVEL SIMULATION OF DIGITAL SYSTEMS USING N.mPc



Jack Gjovaag is a software engineer in the Computer Research Lab, part of the Applied Research Laboratories. Jack joined Tektronix in 1975 from California Computer Products. His experience includes computer graphics, computer aided design, and automated cartography.



Marc Wells is a hardware/software engineer in the Computer Research Lab, part of the Applied Research Laboratories. Marc joined Tektronix in 1974. He has a BA in math and physics from Whitman College, 1974.

With high level simulation, digital system behavior can be studied without completing a full hardware design and implementation, software evaluation can begin before hardware is available, and design changes are more easily made than if the design were implemented in hardware. This article describes the N.mPc simulation system and discusses some cases where it is being used profitably.

The traditional approach to digital system design involves carrying a tentative design down to a point where it can be implemented in hardware, building a prototype of the design, and then evaluating the design by running appropriate software on the prototype hardware. Based upon observed performance, the design may change. These changes must be installed in the prototype hardware and the process repeated. This approach is outlined in figure 1.

It is important to observe that most performance characteristics of a system are determined by the time the functional description of the hardware is specified. Thus building a hardware prototype to investigate performance involves more detail designing than is theoretically necessary. This overdesigning is usually done only because it is difficult to analyze the performance of a complex system based on its functional description.

The N.mPc system is intended to improve the design methodology shown in figure 1 by providing methods for design evaluation before prototyping. The major advantage of this is that the system need not be designed to greater detail than that necessary for validating the system. This is done by providing a formal language that can be used to describe the function of a digital system, some programs that support development of system software, and a simulator that will run the software on the described system and permit observation of performance. The design methodology using N.mPc is shown in figure 2.

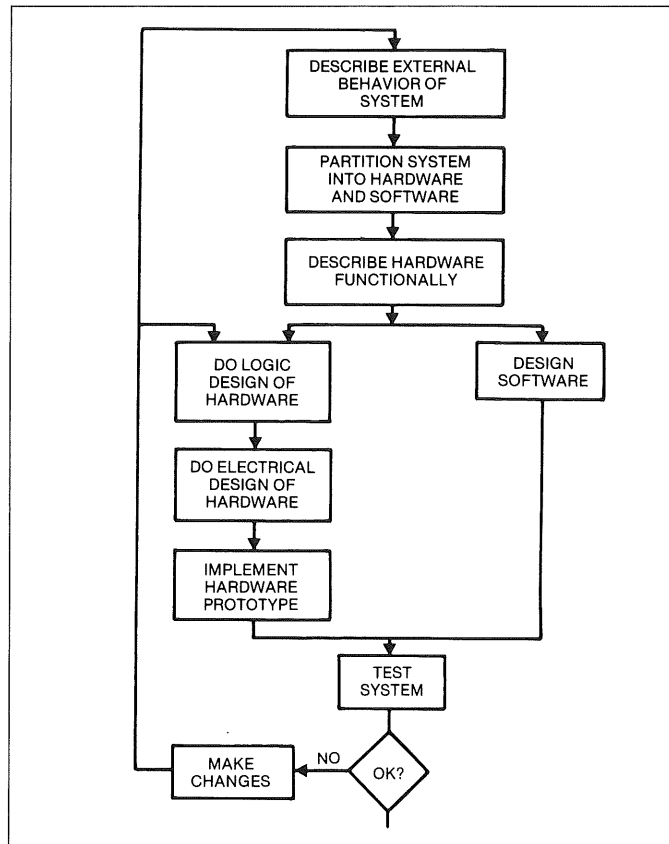


Figure 1. The traditional design process for digital systems requires building a hardware prototype to evaluate the design. N.mPc eliminates the prototyping bottleneck (see figure 2).

Register transfer level descriptions

Note in figure 2 that the design loop is completed before detailed logical and electrical design. For this to be possible, the design must be expressed on a higher level of abstraction than logic gates. The N.mPc system uses a register transfer level (or RT) description to express the function of the system being designed. An RT description is composed of two types of components: registers, which are devices for holding collections of bits; and transfer functions, which alter and transfer data between registers. Such a system description will undergo a sequence of discrete states as fields of bits are transformed by the transfer functions and stored in destination registers. Generally there is a bit or short field of bits (called a clock) that is a product term in most of the transfer functions. This clock field changes in step with time, thereby enabling transfer functions only at discrete time intervals.¹

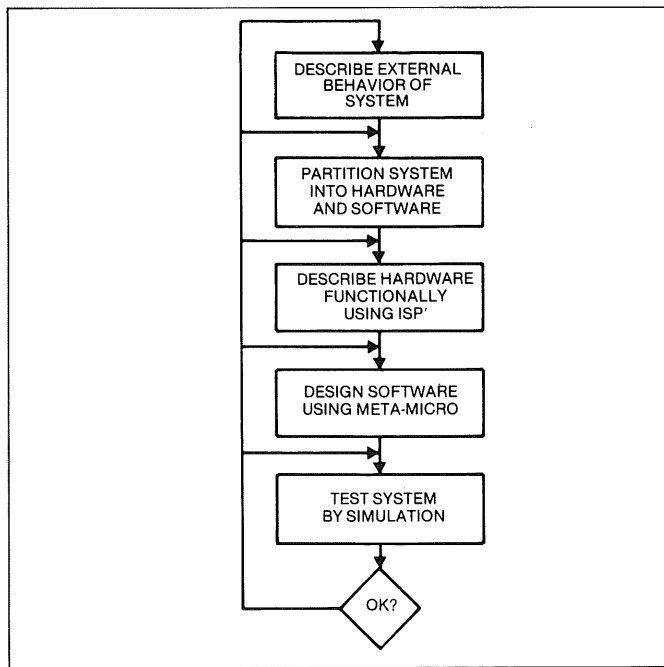


Figure 2. The design process using N.mPc eliminates the early hardware prototype traditionally required for digital system evaluation.

A register transfer description has the power to describe transfer functions at a level of abstraction higher than that of logic gates. For example, an RT statement to transfer the product of two 8-bit registers called A and B to a 16-bit register called C is as follows:

```

state A <8>,
B <8>,
C <16>;
main: =(C = A * B)
  
```

The first three lines merely declare the size and names of the registers and the third defines the single transfer function of our example description. Notice the strong similarity between the example and a small program written in a high level programming language. In fact, the similarity is more than just appearance; RT descriptions may be executed or caused to simulate the behavior of the described system. The contents of the registers, as well as the elapsed simulated time, may be examined during the simulation. Thus the system behavior over time can be accurately predicted.

Components of the N.mPc system

The example RT description shown above is stated in a RT language called ISP', which is a component of the N.mPc system. The complete N.mPc system is shown in figure 3. RT descriptions are processed by the ISP' Compiler. Simulated memories and their initial contents are prepared through the use of a flexible component called the MetaMicro Assembler and the Linking/Loader. Simulated memories and compiled ISP' descriptions are combined according to interconnection information supplied in the ecology file by the Ecologist program to produce a simulation file. The simulation file is an executable file which when run will

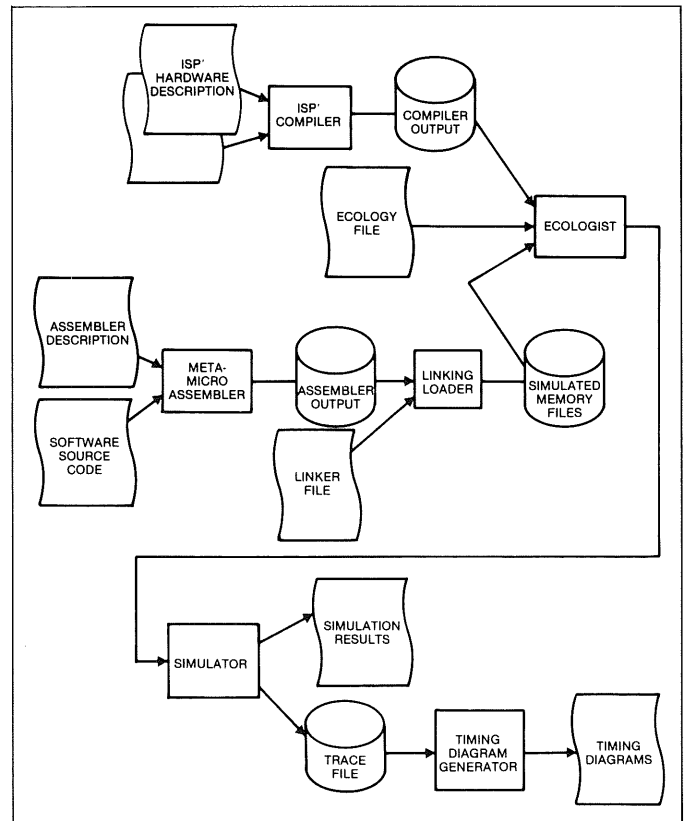


Figure 3. The N.mPc system.

allow interactive examination of the behavior of the system and will gather timing information for selected registers. Graphic timing diagrams can be produced using a postprocessor called td.

ISP' Descriptions

In this section we will give a brief description of the ISP' language and show some examples of its use.

Structures

The registers in an ISP' description store information between transfers. The registers are called *structures* in the documentation. The registers are of three types:

- **States** are registers internal to the ISP' design. They may be declared to be of varying length with varying bit numbering. For example, the declaration

```
state accum <1:9>;
```

declares a 9-bit register called accum whose high-order bit is numbered 1. A file of registers may be declared as follows:

```
state status[0:7]<15:0>;
```
- **Ports** are registers which may be connected externally to other ISP' descriptions. Ports may be written by one ISP' description and read by another, thus implementing communication between modules.
- **Memories** are special arrays of constant-length registers that may be set to a desired initial state through use of the Meta-Micro assembler, the Linking/Loader and the Ecologist.

Register transfer statements

Register transfers are specified with register transfer statements, which correspond roughly to assignment statements in a programming language. For example, the register transfer statement

Areg = Buff *: logical - 3;

will cause the contents of Buff to be logically right shifted by 3 bits and the result transferred to Areg. ISP' provides a broad set of operators for specifying the register transfer function, such as including the common arithmetic and logical operators.

Simulated time

Perhaps the biggest difference between ISP' descriptions and programs written in a conventional high level language is the concept of *simulated time*. Each execution of a register transfer statement has an associated number, which is its simulated time of occurrence. This number orders the transfers in simulated time such that the states produced by a transfer at a certain simulated time can have no effect on transfers at an earlier simulated time. Also, all transfers occurring at the same simulated time are, in effect, simultaneous.

Control of simulated time is achieved with a delay statement. When a delay statement is executed, all transfers prior to it are completed and the simulated time is advanced by the number of time units specified in the statement. The units – microseconds, nanoseconds – are not specified in the ISP' description.

Without specific instructions to the contrary, all transfers with the same simulated time of occurrence will be treated as parallel actions. Thus the contents of two registers can be exchanged by the following pair of transfer statements without destroying one of the values:

```
a = b;
b = a;
```

It is sometimes desirable to impose sequentiality of execution on transfers that occur at the same simulated time. For example, if one wishes to transfer the contents of register r1 to register r2 and then transfer the new contents of r2 to register r3, the first transfer must be forced to complete before the second. This is done with the *next* statement:

```
r2 = r1;
next
r3 = r2;
```

The *next* statement causes all transfers preceding it at the current simulated time to be completed but without advancing the simulated time.

Considerable flexibility in the execution order of statements is provided through a conventional set of control-flow statements such as an *if* statement, a *while* and *do until*, a *case* statement and a *procedure call*.

Processes

Statements and declarations are grouped into collections called processes. Processes are independent of each other in the sense that they can all run concurrently. Each process may be in one of three states:

Running: All conditions for the execution of the process have been satisfied and the process has reached the head of a queue of processes eligible to run.

Ready: All conditions for the execution of the process have been satisfied but the process is not at the head of the queue of processes eligible to run.

Waiting: Some condition necessary for the execution of the process has not been satisfied. The process will sleep until all conditions have been satisfied then its status will be changed to ready.

Conditions necessary for executing a process are established by the process declaration statement and by *wait* and *delay* statements within the process body. A main process is a process that is executed repeatedly during the simulation except when made to wait by *wait* or *delay* statements. A main process is placed in a running status at the beginning of a simulation. A *when* process is a process that becomes ready only when some event on a port occurs. Possible events are leading edge, trailing edge, or change port value. When the last statement of a *when* process is executed, the process sleeps again until the event reoccurs.

The preceding discussion of ISP' is incomplete and serves only to illustrate the major features of the language. It should, however, be sufficient to permit understanding of practical ISP' descriptions. One such description taken from a much larger simulation done in the Computer Research Labs is shown in figure 4. It describes a process that manages the reading and writing of an array of random access memory.

```
/*
 * Memory management process
 */
macro   LONGWORD   =      31:0&,
        .           =      ; next; &;

port    LRW,        !read/write signal
        LUBE,        !upper byte enable
        LLBE,        !lower byte enable
        LAS,         !address strobe
        LDS,         !data strobe
        LAck,        !data acknowledge
        LBus <LONGWORD>; !local bus

state   address <23:1>;
memory  datamem [0:32767] <7:0>;

when (LAS:lead) :=
(
    address = LBus <23:1>;
    delay (120);
    LAck = 1;
    LDS = 1;
    if LRW
    (
        delay (60);
        if LUBE datamem [address *: logical 1] = LBus <15:8>;
        if LLBE datamem [address *: logical 1 + 1] = LBus <7:0>;
    )
    else
    (
        if LUBE LBus <15:8> = datamem [address *: logical 1];
        if LLBE LBus <7:0> = datamem [address *: logical 1 + 1];
    );
    wait (LAS:trail);
    LBus = 0;
    LAck = 0;
    LDS = 0;
)
```

Figure 4. An ISP' description that describes the reading and writing of a random access memory.

Integration of Processes

Compiled ISP' descriptions can be compared to classes of electronic packages with pins for connection to the outside world where the pins correspond to ISP' ports. Integrating these descriptions into a system amounts to selecting as many "packages" of each type as needed and interconnecting them. This interconnection is done with the topology file.

The topology file has two major components: signal declarations – where a signal is a named connection between ports, and processor declarations – where the term *processor* means a compiled ISP' description. Signal declarations occur at the beginning of the topology file and are similar in form to the state and port declarations in ISP'. Processor declarations specify: the ISP' description to be used, a name by which the processor may be identified during simulation, the files containing the initial contents of memories used (described more fully in the next section,) the time units used in delay statements, and the connections between ports in the ISP' description and the declared-signal names.

Signals are the names by which certain states internal to a processor may be referenced, however one processor cannot alter the value of a port in another processor. The value of a signal is the logical sum (OR) of the values of all of the ports connected to the signal. Thus a port may be viewed as an open-collector-output negative-logic transceiver.

Simulated Software

Any system which uses a programmable device (such as a micro-processor or a ROM-based state machine) must have some way of specifying the program for that device. Since N.mPc is intended to be a general purpose tool, it must provide a way of generating programming data for many devices, including commercially available parts as well as custom designed applications. The solution to this programming problem is the MetaMicro Assembler.

The MetaMicro Assembler

Writing a program using the MetaMicro Assembler is a two-stage effort. First, the instruction set of the target machine must be defined and the mapping between instruction mnemonic and machine bit format specified. Second, the actual code has to be written for the target machine. The instruction set is specified in the declaration section of the assembler source file, the actual code is contained in the instruction section. Figure 5 shows a sample of a MetaMicro Assembler declaration, a subset of the Intel 8080 instruction set.

The *instr* declaration specifies the name which will refer to instructions elsewhere in the assembler. The maximum number of machine words per instruction and the default number of machine words per instruction are indicated between square brackets. The number enclosed in angle brackets indicates the number of bits per machine word. The *format* declaration specifies the names of various fields of the instruction. The number in square brackets indicates which word of a multiword instruction is to be used; the numbers in angle brackets indicate which bits of the word are to be used.

Meta-Micro Assembler Declaration Section for 8080

```
!*****
!* i8080.m.
!* metaMicro description file for Intel 8080
!*****

instr  instr[3,1]<8>    $ ! three words of eight bits each
                        ! default length of instruction is 1

format  op      = instr[0]<7:6>, !main op code
        dst     = instr[0]<5:3>, !destination or op code
        src     = instr[0]<2:0>, !source or op code
        rx      = instr[0]<5:4>, !register pair
        wd1     = instr[0]<7:0>, !whole first word
        wd2     = instr[1]<7:0>, !whole second word
        wd3     = instr[2]<7:0>$ !whole third word

macro   ret      = wd1-0311 $    $,      !return unconditional
        rnz      = wd1-0300 $    $,      !return no zero (Z=0)
        rz       = wd1-0310 $    $,      !return zero (Z=1)
        rnc      = wd1-0320 $    $,      !return no carry (CY=0)
        :
        :
        :

!      ***** This section contains the 8080 illegal op codes *****

bind    iopc     "ILLEGAL OP CODE FORMED",
        bcde     "REGISTER PAIR MAY ONLY BE: bc OR de",
        rplc     "REGISTER PAIR MAY ONLY BE: b,d,h,sp",
        rp2c     "REGISTER PAIR MAY ONLY BE: b,d,h,psw",
        mbrg     "OPERAND MUST BE A REGISTER" $

illegal(val)= (wd1=val;){iocp$,      !macro for illegal declarations
illegal (wd1-0010) iopc, (wd1-0020) iopc, (wd1-0030) iopc,
(wd1-0040) iopc, (wd1-0050) iopc, (wd1-0060) iopc,
(wd1-0070) iopc, (wd1-0131) iopc, (wd1-0165) iopc,
(wd1 = 0375) bcde,
(wd1 = 0335) rplc,
(wd1 = 0331) rp2c,
(wd1 = 0355) mbrg $
```

Figure 5. A MetaMicro Assembler declaration. This is a subset of the Intel 8080 instruction set.

Bit patterns are built using macros. The *macro* declaration specifies the way in which bit patterns are to be combined to form machine instructions. Parameters are passed to the macro; these parameters determine the value of bit fields in the completed instruction. Several types of arithmetic and logical operations may be performed on the macro parameters. Conditional expressions of the form *if-then* or *if-then-else* may be used at any time, either as part of a macro or in the instruction section itself, to perform conditional assembly.

The assembler can be set by the *illegal* declaration to indicate when an illegal bit pattern has been formed. When an illegal bit pattern is detected, a user defined warning message is printed.

The instruction section of the assembler is what is converted into bit patterns. If the instruction definition macros have been set up well, code written for the MetaMicro Assembler will look like any ordinary assembler code. In fact, declaration sections for several processors, such as the 8080, Z80, 68000, and so on, have been written which provide a syntax very similar to that of the vendor-supplied assemblers (see figure 6.)

Once the assembler has been defined and the desired code written, the assembler creates an output file which the Linking Loader uses to create a machine executable image of the program.

The Linking Loader

The Linking Loader takes one or more files from the assembler and links them together to fit in the address space specified. Again, since the Linking Loader is designed as a general tool, there are several aspects of the operation which are user definable. Figure 7 is an example of the command file for the Linking Loader.

Meta-Micro Assembler Instruction Section for 8080

```
include /nmpc/softgen/mmpd/i8080$
begin
    lxi (h,128)
    lxi (sp,256)
    lxi (b,700)
loop:
    dcr (c)
    jnz (loop)
    dcr (b)
    jnz (loop)
    hlt
end
```

Figure 6. An example of a program written for the 8080 using the MetaMicro Assembler.

The size and format of the instruction word are specified as in the MetaMicro Assembler using the *instr* and *format* keywords.

The *mode* declaration specifies the method to be used to resolve address references. A particular piece of code, such as a sub-routine, must have its address field modified to reflect the new destination address.

It may be necessary for the linker to break up a sequence of instructions and move some of them to a physically different area in memory. In order to maintain the correct program flow, a jump to the new address must be inserted. The method for doing this is specified by the *transfer* directive.

Linking Loader Directives for 8080

```
!*****!
!* i8080.i *!
!* Linking Loader description for the Intel 8080 *!
!*****!

instr
  inst[3,1]<8>$

format
  op = inst[0]<7:6>,
  dst = inst[0]<5:3>,
  src = inst[0]<2:0>,
  rx = inst[0]<5:4>,
  wd1 = inst[0]<7:0>,
  wd2 = inst[1]<7:0>,
  wd3 = inst[2]<7:0>$

space
  <0:4095>$

transfer
  {new
    wd1 = 0303$
    wd2 = address$
    wd3 = address ^ -8$
    length = 3$
  }

mode
  case length eql 3:
    wd2 ~ (wd3^8) + wd2 + address$
    wd3 ~ ((wd3^8) + wd2 + address)^-8$
    break$
  esac,

  default:
    wd1 ~ (wd2^8) + wd1 + address$
    wd2 ~ ((wd2^8) + wd1 + address)^-8$
    break$
  esac$
```

Figure 7. Linking Loader directives example.

The *space* declaration defines the memory space into which the code will be allocated. The numbers in angle brackets specify the lower and upper address of a block of available memory. Any memory locations not specifically included in a *space* statement are not used and any attempt to store information in these locations will cause an error.

Once the address modification and transfer mechanisms have been defined, the memory allocator can be invoked. There are several user-selectable methods of allocating the memory, including a fragmented scheme, high- or low-end packing and a modified first-fit scheme. The output of the allocator is a file which is ready to be loaded into an N.mPc simulation.

A utility to convert from the N.mPc machine image format to a format readable by the TLOGS gate-level logic simulator has been written.² This allows files for use in TLOGS to be written using the MetaMicro Assembler. It is important to note that the allocator-output file can be used (by an appropriate utility program) to program PROMs for the target hardware. The capability to program PROMs is very valuable because software can be written and debugged using the simulator before any hardware is available.

Running Simulations

Once all the pieces have been put together, from ISP' descriptions of the hardware components to the MetaMicro Assembler descriptions of the software, the simulation is ready to be run. The simulation runtime environment is interactive. The user controls the simulation from a terminal which is used to start and stop the simulation, examine and modify registers in the various processes, and examine the various memories. Data can also be saved in a file which can later be processed to generate timing diagrams.

Displaying and modifying data

Before running the simulation, the parts of the simulation that the user wants to watch should be specified. The value of registers within processes, the value of signals between processes, and the contents of any memory can be monitored throughout the simulation. The *display* command is used to watch registers within processes. When anything selected is written to, or optionally read, its value and the simulation time is printed on the users terminal. The *trace* command is identical to *display* except the output goes to the trace file. In the example below, the value of register *step* in process *stmachine* is printed to the users terminal whereas the value of register *next* of the same process is saved in the trace file:

```
display stmachine:step read
trace stmachine:next
```

The *display* data will also be displayed whenever the register is read.

If a signal between processes is monitored with a *display* or *trace*, only values written to the signal from within the specified process will be shown. Since the actual value of a signal is the OR of all processes connected to that signal, a different command must be used to determine the true value of the signal. The *show* command displays signal values on the terminal and the *dump* command saves the signal value in the trace file.

Whenever the simulation is not running, the value of any registers in any process, the value of any signals between processes, and the contents of any memory can be examined and modified. The *examine* command will display the last value written to a register or a signal in the specified process:

```
examine cpu:ir
examine cpu:addrbus
```

The first example will display the value of register *ir* in process *cpu*, the second example will display the last data written to signal *addrbus* by process *cpu*. Memory is displayed using the *memory* command:

```
memory cpu:rom 0 20
memory cpu:rom prout preud
```

In the first example, the contents of memory *rom* in process *cpu* are displayed starting at location 0 and going through location 20. The second example prints the contents of the same memory from the location specified by label *prout* through the location specified by label *preud*.

Data can be loaded into a register, signal, or memory location using the *deposit* command:

```
deposit 0b01001010 cpu:ir
deposit 0h01fa cpu:addrbus
deposit "?" cpu:rom[125]
```

The first example loads a binary value into register *ir*, the second loads a hexadecimal value into signal *addrbus* and the last example loads the value of the ASCII character "?" into location 125 of memory *rom*.

Runtime commands

The simulation is started by entering the *run* command. Simulation will continue until a breakpoint is reached, an error condition occurs, a deadlock condition occurs, or the user interrupts the system. A deadlock occurs when there are no processes ready to run, that is, every process is waiting for another process and no process is running.

Breakpoints can be set to stop the simulation on the occurrence of some event within the simulation. The value of some register within a process, the simulation time, or a combination of these can be used to form a breakpoint. If, for example, some process named *cpu* contains a register named *pc*, the following will stop the simulation when the value of *pc* is greater than or equal to 5:

```
bkpt cpu:pc geq 5
```

The form:

```
bkpt 500
```

will cause the simulation to stop at simulation time 500.

The following conditions can be monitored as breakpoints: the value of a register is less than, equal to, or greater than a given value; if the value of a register changes either up or down; or if a register is written or read. Breakpoints may be linked so that

complicated relationships may be specified. The breakpoint specification:

```
bkpt 250 after proc1: ir eq 0h1f
```

will stop simulation 250 time steps after register *ir* in process *proc1* is equal to hexadecimal 1f.

N.mPc has many more commands and combinations of commands available than those outlined above. There are methods for making timing measurements, for determining the minimum and maximum values of registers, and for triggering events based on the state of the simulation. Any of these commands may be combined in a file that, when loaded, will automatically execute the commands in the file.

Creating timing diagrams

A program, named *td*, to draw timing diagrams from data saved in the trace file by the *trace* and *dump* commands has been written.³ The timing diagrams are displayed using printer characters so that no special graphics capability is needed; timing diagrams can be displayed on the same terminal used to run N.mPc. All these are selectable: the data to be displayed, the order in which it is to be displayed, and the simulation time period over which it is to be displayed. Since registers and signals in N.mPc may be made up of more than one bit, the *td* program is capable of displaying multibit data.

A line of data to be displayed is defined by specifying the name of the register or signal, the first bit to display, the number of bits to display and, if more than one bit is being drawn, the radix in which to print the display. Single-bit displays are printed more or less as a conventional timing diagram, multibit displays are printed with the value of the specified bits separated by characters to indicate where transitions occur. Figure 8 is an example of a timing diagram created by the *td* program from data generated by an N.mPc simulation.

Applying N.mPc

N.mPc has been successfully applied to a logic design task in the Computer Research Lab. The task was to redesign a significant portion from a TTL-implemented design into a large scale integrated circuit. While the way that N.mPc was used in this situation was not exactly as its designers had intended, it nevertheless performed an important function that would have otherwise been difficult.

The circuit to be redesigned was documented primarily by schematic diagrams and some supporting descriptions of bus signals. After determining what parts of the original circuit were candidates for LSI implementation, it was necessary to discover the behavior of the signals forming its interface. For this purpose, the entire circuit was described in ISP', sometimes using quite low-level transfer functions that modeled the individual logic gates when the functionality of a part of the circuit was not well understood.

When the description was completed and simulations run, the behavior of the circuit could be studied and its function better understood. While it was not necessary, low level descriptions were replaced by higher level constructs as our understanding improved. Once the simulation was running correctly and we comprehended its workings, the part to be redesigned was described in ISP' and the functions to be replaced were removed.

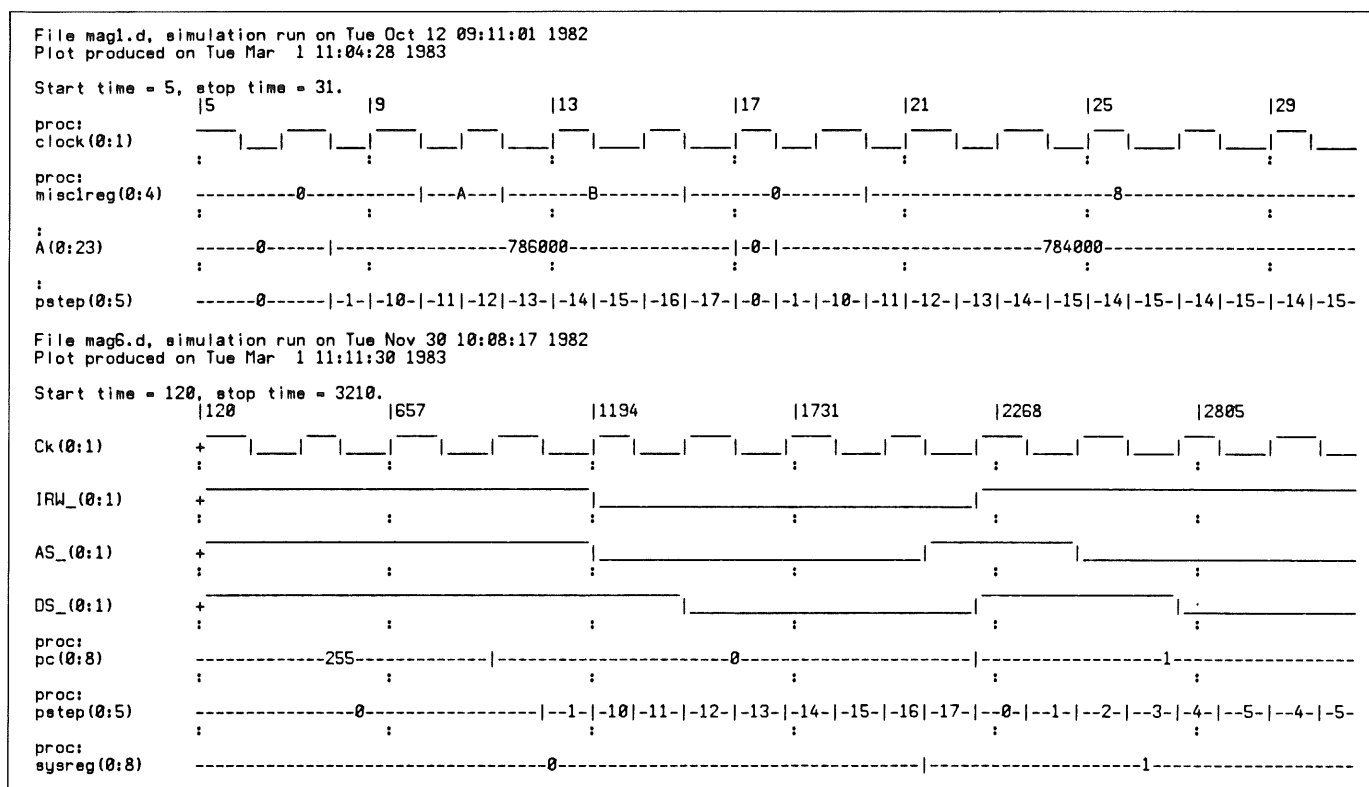


Figure 8. A timing diagram produced by td from N.mPc output.

The new description was connected to what remained of the old and the simulation was rerun to validate the new design.

The value of this simulation was demonstrated when a number of design errors were discovered that might have gone unnoticed through chip fabrication. Obviously, when the circuit being designed is a chip, rework is not possible.

Another project is being planned where N.mPc is expected to be valuable. A large array of very simple processors organized in a tree structure is planned to be implemented in VLSI. The expected benefits from simulating the chip at the register transfer level before constructing it are:

- Determine the best workload distribution among the processors.
- Minimize the required communication bandwidth between processors.
- Establish effective interprocessor communication protocols.
- Measure the overall performance of the system.
- Encourage experimentation early in the design.

Conclusions

The high level digital simulation system N.mPc has proved to be quite useful in system design. We feel that in one project, it saved considerable effort and helped locate errors that might have otherwise been missed. At least one advanced application of N.mPc is planned where even greater benefits are anticipated.

One note of caution is in order. The N.mPc system was designed and developed in a noncommercial research environment. The syntax of the various languages used – ISP', topology description, MetaMicro and the Linking Loader – are often inconsistent for no apparent reason. Some documented features have been discovered to be unimplemented. The system occasionally fails in mysterious ways due to program errors or insufficient error checking. Despite these problems, however, the system is quite workable and effective.

For More Information

For more information, contact Jack Gjovaag ext. B-6160, or Marc Wells ext. B-6179.

Substantial improvements and modifications are being made to N.mPc in another organization. Ellen Mickanin, ext. WR-1909 or Pat Thompson, ext. WR-1006, can elaborate on these changes.

Contact Marc Wells, ext. 6179, or Karen Conrad, ext. B-6155, for copies of the ARG Technical Reports listed below. □

References

1. Bell, C.G., Newell, A., *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.
2. Wells, Marc, *Converting N.mPc Core Image File Format to TLOGS ROM Statement Format*, ARG Technical Report CR-83-3, 1983.
3. Wells, Marc, *Drawing Timing Diagrams from an N.mPc Simulation*, ARG Technical Report CR-83-1, 1983.

ENGINEERING ACTIVITIES COUNCIL MEMBERSHIP CHANGES

The semiannual EAC "rotation" took place recently. About every six months the membership of the Engineering Activities Council (EAC) is partially changed (rotated) as old members finish their terms and new members are selected.

The Engineering Activities Council (EAC) is a group of about 20 engineers and scientists, chartered to stimulate communication between engineering and management, and also among engineers. Members are nominated by their managers, peers, or themselves, and then selected by Bill Walker, executive vice-president, to represent the different engineering disciplines and organizations within Tektronix. The EAC addresses issues of engineering concern through a diverse set of activities including technical forums, technical seminars, new engineer orientations, and engineering surveys.

EAC Goals

- Promote communications between engineers concerning issues such as professional development, new technologies, and internal developments.
- Promote communications from engineering to management concerning issues such as technology, environment, and marketplace trends and pressures. To this end, the EAC needs to be aware of engineers' views and be capable of advising on issues concerning the engineering community.
- Promote communication from management to engineering concerning company philosophy and business directions. □

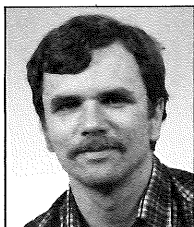


The Engineering Activities Council (EAC). From left to right, front row: Kathy Dagostino, Paul Dittman, Preston Seu, Chairman. Second row: Phil Baker, Pat Green, Tom Ruttan, Nick Fkiaras, Paula Mossaides, Mike Nakamura, Ward Cunningham. Back row: Bill Wilke, Russ Anderson, Mike Hatch, Steve Lyford. Not present: Mark DeSpain, Gary Fladstol, Richard Greco, Geoff Herrick, Bill Trent, Mike Zuhl.

EAC ROSTER

Russ Anderson DAID-DAD/Logic Analyzers	WR-1869	92/726	Mike Hatch I&TG/Portables	DR-2649	39/204
Phil Baker I&TG/Solid State Group	DR-3149	13-035	Geoff Herrick I&TG/Solid State Group	DR-6374	59/840
Ward Cunningham I&TG/Applied Research Group	DR-6180	50/384	Steve Lyford I&TG/Portable Instruments	DR-2952	39/194
Kathy Dagostino DAID-DAD/MDP	WR-1729	92/515	Paula Mossaides DAID-IDD/ECS	W1-2352	61/215
Mark De Spain DAID-IDD/GPP	WL-3755	63/356	Mike Nakamura C&IG/TV Products	DR-1343	58/594
Paul Dittman I&TG-ISD/Lab Instruments	DR-3058	39/111	Tom Ruttan C&IG/FDI	DR-1463	58/733
Nick Fkiaras I&TG/Computer-Aided Engineering	WL-3033	63/397	Preston Seu DAID-DAD/GPP	W1-3856	63/356
Gary Fladstol I&TG-ISD/Lab Instruments	DR-3064	39/103	Bill Trent C&IG/CNA	DR-1447	58/305
Richard Greco DAID-IDD/GAS	W1-3176	63/196	Bill Wilke I&TG/GPI	WR-1521	92/815
Pat Green I&TG/Display Group	DR-5461	50/252	Mike Zuhl DAID-IDD/ECS	W1-2551	61/215

MODULA-2 AS A SOFTWARE ENGINEERING TOOL



Patrick Clancy is a software engineer II in the Advanced Instrument Research Group, part of the Applied Research Group. Pat joined Tektronix in 1981. He received his MSCS from the University of Wisconsin, Madison.

Modula-2 is a programming language developed by Niklaus Wirth and is a descendent of Pascal. Because it is intended for systems programming, Modula-2 provides many features not found in Pascal, or even in Pascal extensions. These features enhance Modula-2's suitability for the creation of modular systems, for providing access to the underlying machine, and for real-time control applications. These are the areas in which Pascal is most deficient, and where the need for a next-generation Pascal successor has been most strongly felt by systems programmers.

This article reports on the construction of a Modula-2 compiler and run-time system for the Motorola MC68000 microprocessor, and on the desirable language features which led to the undertaking of this project. This work was done as part of a program in Advanced Instrument Research to develop computer-based

instrumentation. Using the Modula-2 tools, a software development team has created stand-alone modules to drive the hardware of an instrument system prototype.

Modula-2 was chosen over more readily available languages because it was better suited to meet the stringent requirements of software engineering for complex instrument systems. The most important requirements we identified were (1) modularity of design, (2) reconfigurability of the system, (3) compiler type-checking across all system interfaces, and (4) the ability to do all coding in the high-level language (HLL); assembly code should not be required. (Assembly code can be included, but it is not required as it is in other languages for certain functions.)

Language Features

A brief overview of Modula-2's distinguishing features will be presented here. A manual and report on Modula-2 can be found in [8].

The basic control and data structures of Modula-2 and Pascal are almost identical. (Modula-2 has one control construct, LOOP/EXIT, and two data types, procedure and process, which do not have Pascal counterparts). In this article, Pascal will be used as the basis for comparison.

Modules

Programs in Modula-2 are written in terms of modules. All separate compilation units are modules; in addition, modules may be nested within procedures or other modules. Modules are a means of partitioning a program and creating abstract data types. Modules can be used to group together related operations and hide global data that should not be generally accessible.

The most significant aspect of the module feature is its role in providing a separate compilation capability within the language. A special type of module, called the *definition module*, is used as an interface between compilation units. The information from definition modules is accessed by the compiler to type check externally referenced (imported) names. This type checking is as complete as the checking within compilation units, since declarations in definition modules provide all the necessary information. For example, complete checking of parameter types is done on the usage of procedures exported from separately compiled modules. This capability is usually unavailable in Pascal, including those extended Pascals that provide for separate compilation, since major syntactic modifications and additions would be necessary.

Each definition module has a corresponding *implementation module*, which provides the actual code body of all procedures declared in the definition module. In addition, the implementation module may contain other objects which are not exported. The compiler checks the consistency between declarations in the definition module and their corresponding implementations. The important consequence of this mechanism is that the implementation in one module may be changed with certain knowledge that other parts of the system will continue to function correctly as long as the interface (definition module) remains unchanged. The interfaces are the anchor points for the overall program design, allowing the implementations to be carried out independently, possibly by different members of a software design team. The overall structure of a Modula-2 program is shown in figure 1.

Processes and Interrupts

The real-time control requirements that characterize complex hardware systems usually cannot be met within the framework of commonly used HLLs. For this reason, the software controlling such systems often contains some assembly code to deal with machine-level tasks such as interrupt-handling. Modula-2 provides the facilities to program real-time control entirely within the language. This is done without enforcing a particular high-level view of processes and scheduling, as is found in other multiprocessing languages, such as Concurrent Pascal [3], Mesa [5], or Ada [6].

To support multiprocessing, Modula-2 provides a *PROCESS* data type and a *TRANSFER* operation on this type to effect a context switch. This constitutes a simple co-routine mechanism. A scheduler module (written in Modula-2) can be provided to implement true multitasking. Modula-2 thus provides the lowest-common-denominator mechanism upon which any sort of multitasking can be easily built; a scheduler can be constructed in about 50 lines of Modula-2 code.

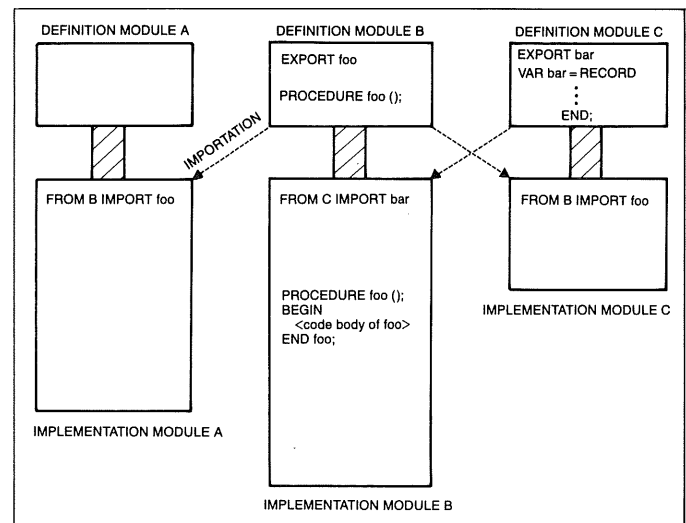


Figure 1. The structure of a Modula-2 program. All exports out of the compilation unit are from definition modules. Definition modules may contain all types of declarations, but no executable statements. For exported procedures, the procedure heading only appears in the definition module. If a definition module remains fixed, the corresponding implementation module can be changed at will without affecting the correct functioning of other parts of the system.

To support interrupt-handling, Modula-2 provides an *IOTRANSFER* operation. This operation is similar to *TRANSFER* except that it causes the caller to suspend pending a specified interrupt. Interrupt handlers are therefore written in Modula-2. A priority specification for modules is also provided, so that interrupts having a given hardware priority may be disabled during execution of critical code. This establishes a "critical section" [2] within a module, which guarantees that operations on data accessible to multiple processes will be indivisible and correct. Once again, Modula-2 provides a simple and elegant abstraction of the machine architecture, which allows complete flexibility in implementing higher-level operating system functions (see figure 2).

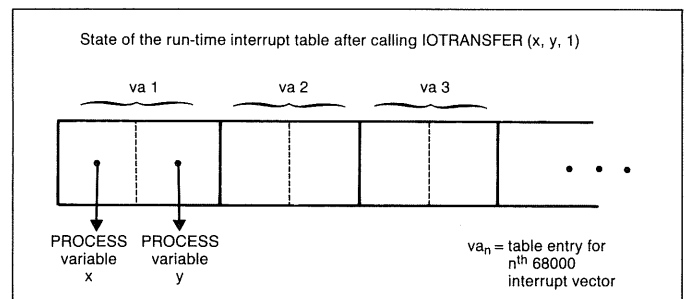


Figure 2. To support interrupt-handling, Modula-2 employs an operation called *IOTRANSFER*. This operation causes the caller to suspend pending a specified interrupt. The effect of calling *IOTRANSFER* (X,Y,1) is shown. When interrupt 1 occurs, the current process state is stored in Y and process X is activated.

Although these system operations were originally modeled after the PDP-11 processor architecture [8], we found that they were

entirely sufficient for the 68000 as well. They are probably a good fit with most processors that have an interrupt capability.

Machine Access

One reason assembly language has commonly been used in system-level programming is the insulation of many HLLs from access to the underlying machine. Such insulation restricts the system-programmer in languages—like Pascal—that enforce strong type checking rules.

Modula-2 uses Pascal-style type checking but provides methods to circumvent the checks. This is done by using special unchecked data types (WORD, ADDRESS), or using type names as type-transfer functions. It is still up to the programmer to isolate hardware-dependent code into modules that can be easily identified and replaced when hardware is modified.

The Compiler

The complete compiler system was created by constructing a front-end compiler using UNIX* tools (C, yacc, lex), and incorporating this compiler with an existing code generator for the 68000. The code generator comes from the GCS Pascal compiler, which was discussed in a previous issue of *Technology Report* [7]. Because of the close similarity of most Modula-2 and Pascal statement and data types, the intermediate code form required as input to the code generator was with few exceptions sufficient as an output language for the front-end. (The intermediate code consists of quadruples, which resemble a high-level assembly language.)

The front-end compiler operates in two passes. Two passes are required because Modula-2 allows references within the same scope to identifiers whose declarations have not yet been encountered. Thus, the forward declaration of Pascal is not required, and constant/type/variable declarations need not be grouped at the beginning of a block.

In addition to the usual syntactic and semantic checks, the information from definition modules must be accessed for externally referenced (imported) names. This accessing of symbol-table information from other compilations (completed during the first pass) constitutes, in fact, a separate linking step. This step is not found in languages that have no interface specifications. The compiler does version control when accessing this external information, by checking time stamps created when the definition modules were compiled; this ensures that imports of the same name are accessing identical information. (Different versions bearing the same name would be created if, for example, a definition module were modified and recompiled in between compilations of two other importing modules. This situation is usually to be avoided). This version-control process is illustrated in figure 3.

Run-Time System

The run-time system must implement the operations on the PROCESS data type, in addition to the more usual operations such as error checking and floating-point processing. Because of the simplicity of the system model, the amount of extra run-time support code for PROCESS operations is relatively small.

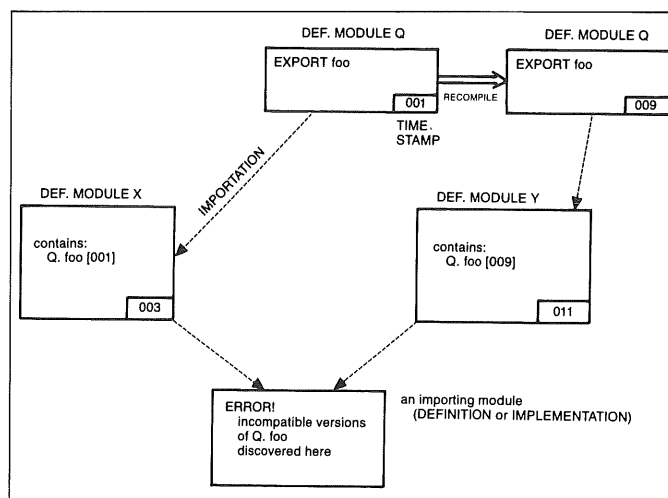


Figure 3. In the Modula-2 compiler, version control is accomplished by checking time stamps. This ensures that all imports bearing the same version name are identical.

One complication arises from the dynamic nature of the interrupt specification; a particular interrupt vector on the 68000 can be dynamically associated with different sets of Modula-2 processes. This association requires that an interrupt table exist at run-time, to store pointers to the process state variables (see figure 2).

We have written an operating-system kernel in Modula-2 to provide multitasking (and eventually multiprocessor) capabilities, as well as various operating-system utilities such as a terminal driver. This kernel constitutes a stand-alone system for the 68000 single-board computers that control instrument system hardware.

Experiences with Modula-2

Modula-2 has met the requirements of modularity, reconfigurability, inter-module type checking, and real-time control capability that are crucial to well-engineered software for a complex computer-based instrument system. In fact, we expect the benefits of using Modula-2 will increase as the size and complexity of software development efforts increase—because interface specification and modularity become correspondingly more important under these circumstances.

Problems and Qualifications

While this article has emphasized Modula-2's good points, there are also some qualifications to be made, and problems to be pointed out:

1. Some language features need to be either clarified, modified, or extended. In order to write flexible I/O functions, for example, variable argument counts in procedure calls should be allowed in some situations. Other problems include the lack of a way to specify initial values in variable declarations and Modula-2's overly restrictive limitation on set size.
2. Modula-2 is harder to compile than Pascal. One reason for this is the greater sophistication of the scope-of-visibility rules for identifiers—this sophistication is due to the module feature. These rules make compiler symbol-table operations much more complex than those needed for Pascal. This extra

*UNIX is a trademark of Bell Laboratories.

degree of flexibility in controlling identifier visibility is termed scope control [1], and implementation solutions have not yet been much discussed in the compiler literature.

The definition-module interface also adds to compiler complexity. A Modula-2 compiler therefore requires more machine resources to run than a Pascal compiler. (The compiler developed by the author runs on a VAX 11/780.) Note, however, that Modula-2 presents fewer implementation problems (because of its inherent simplicity) than some other systems-programming languages, such as Ada.

3. While Modula-2 can facilitate the software engineering of large programming projects, it is not necessarily the best choice for doing a small job fast. This is because module interfacing and type checking add overhead to the program development process; for very small programs the benefits which go along with this overhead will not be apparent.

Conclusions

Modula-2 provides an excellent basis for applying software engineering principles to large software projects, especially for operating-system and stand-alone systems development. As computer-based systems become more complex, high-level language features that can manage complexity will be increasingly important in the software development process. □

References

- [1] R.P. Cook and T.J. LeBlanc, "A Symbol Table Abstraction to Implement Languages with Explicit Scope Control," *IEEE Trans. Software Eng.*, Jan. 1983.
- [2] Dijkstra, E.W., "Cooperating Sequential Processes," in F. Genuys (ed.), *Programming Languages*, Academic Press, 1968.
- [3] Brinch Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Trans. Software Eng.*, June 1975, p. 199-207.
- [4] Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report, Second Edition*, Springer-Verlag, 1974.
- [5] Mitchell, J.G., Maybury, W., and Sweet, R., *Mesa Language Manual*, Xerox Res. Ctr., Palo Alto, Calif., 1979.
- [6] *Reference Manual for the Ada Programming Language, Proposed Standard Document*, United States Department of Defense, July 1980, GPO 008-000-00354-8.
- [7] Allen Wirfs-Brock and Paul McCullough, "A Pascal Compiler for Motorola 68000 Firmware Development," *Technology Report*, Sept. 1981.
- [8] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, 1982.

BOOK BY TEK AUTHOR DEALS WITH STANDARDS



Charles D. Sullivan

Charles D. Sullivan, late manager, Technical Standards, has written a monograph *Standards and Standardization*, published by Marcel Dekker, Inc., New York. Chuck has compiled a broad overview of the subject based on his many years of on-the-job involvement with standards.

Standards and Standardization is an excellent introduction to this important subject. Engineering students will welcome it as an essential text that can be studied without prerequisites, and engineers will appreciate the perspective this book offers. It is

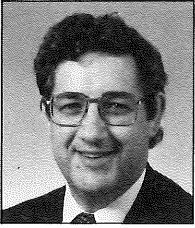
equally valuable for professional seminars and in-house training programs, as well as supplementary reading in social science courses.

Some subjects covered:

- Approaches to standards
- Ancient standards
- Voluntary and mandatory standards
- Mechanics of standards preparation
- Organizations
- The Standards Engineering Society
- Standards coordination
- The General Agreement on Tariffs and Trade (GATT)

For more information, call Bonnie Kookan, ext. B-1800. □

ERGONOMIC BARRIERS TO SALES ARE REAL



Gene Lynch is a human factors engineer in the Systems Engineering group, part of IDD. Gene joined Tektronix in 1981 after teaching math, physics, and mechanical engineering at the Santa Catalina School and the Naval Postgraduate School in Monterey, California. Each summer from 1973 to 1980, he was a consultant to Tek. Gene holds a BS, MS, and PhD in Engineering Science from the University of Notre Dame.

Late in 1981 IDD started losing sales in ergonomically sensitive markets. Sales statistics dramatically pointed toward worse losses to come. To separate ergonomic fact from fiction IDD formed the Human Factors Task Force. Its mission was to establish good communications with our people in the ergonomically sensitive markets and with technical organizations, government commissions, and the standards bodies. Then, the task force was to determine what was needed to eliminate ergonomic barriers to sales. The members of the task force were Bruce Carroci, Bob Edge, Jerry Murch, Bob Russell, and Gene Lynch (chairman).

A little more than two years ago few of us had heard of ergonomics. It was a term that was popular in Europe. Its simplest meaning is the science of making work easier. A more complete definition is the adaptation of equipment and the work environment to meet man's strengths, capabilities, and limitations. Ergonomics is derived from two Greek words: *ergon*, for work and *nemein*, meaning to manage, divide, or distribute.

A World-Wide Challenge

In Europe ergonomics has focused on improving the nature of work with productivity a secondary goal. In the United States the primary interest has been in increasing productivity. Preliminary data supports the position that good ergonomics is good economics.

Germany has two, somewhat conflicting, sets of standards: the German Safety Standard by the TCA (Trade Cooperative Association) and the DIN 66234 by the Deutsches Institut für Normung.

The TCA can, in essence, ban from sale in Germany any equipment violating TCA standards. In Germany it is not uncommon for sales engineers to be met by a purchasing agent holding the pink booklet containing the TCA safety standard. It also is common for workers' councils to have a say in a purchase or in purchase policy. These councils are demanding ergonomically designed terminals and workstations.

The effects of a TCA ban extend beyond Germany. Although only Germany and Sweden have official ergonomic standards, in the absence of an International Standards Association (ISO) standard, most of Europe has accepted the German standards as de facto. The ISO will begin working on a standard this spring.

In Germany it is not uncommon for purchasing agents to greet sales engineers with the TCA safety standard in their hand.

Canada too is active in ergonomics, trying to regulate video display terminals (VDTs). The Task Force on Micro-electronics and Employment has just released recommendations for such regulations in their report, "In the Chips." British Columbia wrote non-binding guidelines last year. This year, the British Columbia Government Employees Union is saying, "the guidelines must be followed."

In the US Too

In this country several states have considered VDT regulations. This is unsettling, because the absence of a national standard could put Tektronix in the untenable position of having to deal with numerous conflicting standards in the American market. The lack of a U.S. position also limits Tek input to the ISO committee that will draft the ISO standard.

Tektronix is working with the Human Factors Society in trying to get an American standard through the American National Standards Institute (ANSI). ANSI is a member of ISO.

Although we are not enthusiastic that ergonomics is an area that needs to be standardized, we strongly support the development of an American position—we face the reality of the German standards, the German standards influence on the ISO, and the development of state-by-state ergonomic standards. We feel that now is the time to pursue a reasonable standard while we have a chance to help define its structure and content.

How does all this affect our products? Without serious attention to ergonomic issues, we could be maneuvered out of one or more markets. We need to develop cost-effective answers to the challenges of this rapidly moving market requirement. *Rather than just meeting the standard, our goal should be to meet our customers needs by following the underlying principles of ergonomics.*

Guidelines Available

Rather than merely designing products to meet the letter of the regulations, IDD people have been proactive in industry efforts to develop standards and guidelines that assure product saleability. One effort, the Human Factors Task Force, has recently published *Guidelines for Eliminating Ergonomic Sales Barriers*.

The Guidelines were developed after closely scrutinizing ergonomic regulations and practices and noting market sensitivities. IDD designers who follow the guidelines should be able to not only avoid building unsaleable products, but their products should be more saleable because the needs of the user are dominant in the Guideline.

Although the Guidelines are based on IDD product and market needs, designers in the other divisions may find the contents of this 17-page publication useful. The guidelines deal with displays, keyboard electronics, workstations, systems considerations, software and measuring techniques. Copies are available from Gene Lynch, d.s. 63-225.

IDD designers may want to discuss specific standards and markets with either Jerry Murch, W1-3858, or Gene Lynch, W1-3730.

Here are some of the highlights from this document:

Displays

The minimum contrast ratio should be 3:1 in normal lighting conditions.

The contrast must be manually adjustable by the operator.

Some sort of antiglare treatment is required.

The display should be flicker free.

Jitter is to be less than 0.7 minutes of arc as seen at 500 mm or less than 0.1 mm actual motion.

Distortions are to be less than ± 1 percent of screen height or width.

Visible variation of brightness across the screen is to be avoided.

The screen surround as well as other surfaces are to have matt finish and with reflectivity between 20 percent to 50 percent.

The display should be adjustable for tilt and viewing distance (integral or optional device).

Characters

A 7x9 matrix is the preferred minimum with a 5x7 as an absolute minimum.

The character height is to be a minimum of 2.6 mm (from the German standards) to a minimum of 3.1 mm (from the Canadian requirements).

Keyboards

Keyboards are to be detached and as thin as possible (30 mm at the home row).

Tactile feedback is desirable, as is optional audio feedback (popular with the French).

When numeric entry is used extensively that a 10 key pad is recommended. If possible, it should be relocatable to the left or right of the main keyboard.



From DESIGN, May 1980

Keyboard slopes should be adjustable (8 degrees for fixed slope keyboards).

Slide stops should be provided to keep the keyboard from slipping on the work surface.

National keyboards should have all legends in the native language.

Red indicators should only indicate warnings, not status.

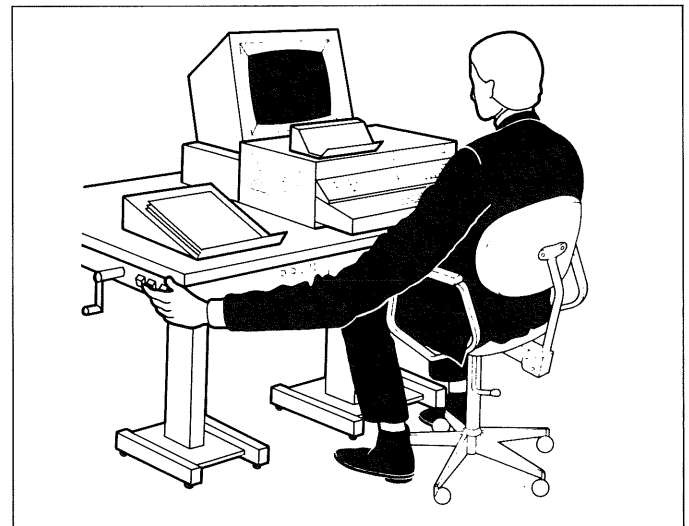
The keyboard cable should allow placing the keyboard up to 6 feet from the center of the display.

The legends on the keyboard should be dark characters on a light background.

System

Controls should be easy to reach and operate.

Equipment plus ambient noise must not exceed 55 dBA. The equipment itself should not exceed 50 dBA (measured at the operator's position).



From DESIGN, May 1980

Thermal comfort is a concern. Exposed surfaces are to be comfortable to the touch. Exposed heat sinks should be kept below 60 degrees C. Hot spots should be avoided. No part of the operator's body should be exposed to a temperature higher than 3 degrees C above the ambient.

Drafts should be avoided. Air flow at the neck, wrists, and ankles should be less than 0.1 meter per second.

The height of the display at its lowest position should be no higher than 14 inches above the work surface. Its highest position should be at least 20 inches above the work surface.

Workstation

Displays should be able to tilt, swivel, and be easily positioned in the work area (by integral design or auxiliary device).

The viewing angle should allow for maximum perpendicularity. Perpendicularity and glare requirements can conflict, so glare control and tilt are both required.

There should be 50 to 100 mm unused space in front of the keyboard on the work surface for a palm rest.

Surfaces should be adjustable to accommodate the range of all persons from those larger than the 5th percent female to smaller than the 95 percent male.

All adjustments should be easy and require little force and should not accidentally readjust.

All corners and edges should be rounded.

There should be adequate leg and knee room.

The elements of the workstation should be modular, reconfigurable, and flexible.

Adequate work surface should be provided for the application.

The workstation should include a proper chair and, where required, a document holder.

Cables should not interfere with the operator.

Software

In the near future (now if possible) system and error messages should be in the native language to match the national keyboards.

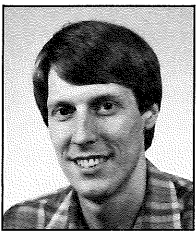
The software should be in the native language.

While standards and ergonomic barriers have not been developed or identified in the area of firmware and software, it is clear that the ergonomic aspects of firmware and software is every bit as important or more important than the physical ergonomics. A DIN committee is working on a standard dealing with man computer dialogues.

For More Information

The Human Factors Task Force has completed its work, but the job of monitoring, tracking and influencing the standards continues. Gene Lynch (63-225/W1-3730) and Jerry Murch (63-489/W1-3858) will update the Guidelines as necessary. They will also be happy to answer questions concerning standards and ergonomics. □

USER INTERFACE ASPECTS OF A DESKTOP CAD SYSTEM



John H. Harms is a software engineer in Graphic Design Application Systems (GDAS) part of IDD. John joined Tektronix in 1980 to work on the team of engineers that designed and implemented Tek 2-D Drafting. He is now working on new CAD systems. John received his BS in computer science from Oregon State University. While at OSU he designed and implemented circuit board placement and routing systems.

The user interface is a very important, but sometimes overlooked, facet of a CAD (computer-aided design/drafting) system. The man-machine interaction needed to make all of the features work effectively really determines a system's usefulness. It is most desirable to have a user interface that makes the system easy to learn and use, and yet provides sufficient power to fully control a complex CAD system. This article details some guidelines for the design or selection of systems that fulfill these goals.

In designing a system that is both easy to use and powerful, the use of a desktop computer in the CAD system has several advantages that make it an attractive alternative to the typical host-

computer terminal configuration. A desktop computer often has display and interaction features that are unavailable on host-based workstations in the same price range. For example, special-function keys may require an expensive intelligent terminal supported by special software. Many desktop computers, however, include such capabilities as standard features.

Other advantages of a desktop computer include instant response (the computer has only one user to think about) and dedicated peripherals (such as plotters and graphic tablets connected directly to the workstation). These advantages make it possible to design a highly interactive user interface at much lower cost than would be possible with a larger computer.

This article examines several facets of man-machine interaction as they relate to drafting systems. Some examples are taken from the Tektronix, Inc. PLOT 50 2-D Drafting package, a computer-aided drafting system that runs on a desktop computer. Four important aspects of a user interface will be discussed:

1. Employing the computer display effectively.
2. Making it easy to enter information into the system.
3. Providing capabilities that allow fast and efficient operation.
4. Preventing unpleasant surprises and uncertainty.

Although the Tektronix 2-D Drafting system is used for examples, the concepts in this article apply to the design of any CAD user interface on a desktop computer.

Employing the Display Effectively

The effectiveness of a user interface depends a great deal on how well the hardware is utilized. In particular, the computer's display is the focal point of the user's attention.

In general, the larger the screen, the better the user interface. For one thing, a large screen has more room for tutorial messages to help the user decide how best to respond. For example, a menu can use full English phrases:

Select dimension alignment

1. Aligned (text at appropriate angle)
2. Unidirectional (text forced horizontal)

With a small screen, it may be necessary to use cryptic abbreviations, like ALIN and UNDR. Such abbreviations tend to increase learning time and force users to refer to manuals more often.

Another advantage of a large, high-resolution screen is the big, detailed image. The user can put more of the drawing on the screen and yet keep small features visible.

Employing a DVST can complicate the design of a good user interface, because once an image is drawn on the screen, it remains until the whole screen is erased. Prompts and messages can quickly fill the screen forcing the image to be redrawn, a process which can take time. Until recently, the only way to get the screen size and resolution of the DVST, without the redraw limitation, was to employ a very expensive high-resolution refresh-display terminal. However, it is now possible to use a relatively inexpensive desktop computer (such as the Tektronix 4054) that provides both stored images and refreshed images. Such refreshed graphics are called "dynamic graphics."

Why are dynamic graphics important? First of all, they go a long way toward eliminating a major objection to DVST displays – the need for redraws. With all messages and user inputs in refresh, there are no full-screen redraws forced by a message area becoming filled.

Since any graphic image can be placed in refresh and moved around on the screen at will, refresh has many other uses. For example, the user can "drag" symbols and text around on the screen until their locations are satisfactory. A complex and accurate cross-hair cursor helps the user precisely locate graphic positions. Other powerful tools such as rubber-band lines (lines that "stretch" as the cursor moves), blinking location indicators, and blinking messages are provided. There is an option for the refresh images to appear in orange, which provides a sharp contrast to the stored green image.

Making It Easy to Enter Information

A system that is easy to learn and use is achieved by designing the user interface to be friendly, yet powerful. This presents the designer with two conflicting goals: simple enough for the novice, but still efficient for the expert.

There are several common methods for telling a CAD system what to do next. One is *command entry*. Here the user types in a command, usually with parameters. For example: COPYROT 45,7 might mean to make seven copies after rotating the image 45°. Although very efficient once learned, the command-entry method requires the user either to memorize command names and parameter orders or to refer repeatedly to a manual. The command method requires extensive user training.

Another common technique is called *menu hierarchy*. This method presents a menu of commands; each command choice from a menu can produce another menu, and so on (see figure 1). In the copy-rotation example, the user might choose EDIT from a menu consisting of ANNOTATION, GEOMETRY, EDIT, and PLOT. His choice might then produce an "edit" menu consisting of BLANK, COPY, MODIFY, DELETE. Selecting COPY would produce a "copy" menu: MIRROR, ROTATE, RESCALE, and TRANSLATE. After ROTATE is selected, the user is asked for the degrees of rotation and the number of copies. Although easier to use than command entry, the menu method can be tedious for an experienced user, who, instead of going directly to the desired function, has to go through many levels of menus.

MENU HIERARCHY EXAMPLE:

ANNOTATION,GEOMETRY,EDIT,PLOT

BLANK,COPY,MODIFY,DELETE

MIRROR,ROTATE,RESCALE,TRANSLATE

ADDITIONAL PROMPTS

Figure 1. The menu-hierarchy technique presents a choice of commands. If the user selects "Edit" from one level, the next level is presented. If "Copy" is selected, a third level appears . . . and so forth.

A third method, which is gaining popularity, uses the *tablet menu*. The tablet menu is a formatted menu on a digitizing tablet from which the user selects by pointing with a special pen (see figure 2). In the copy-rotate example, the user points to the COPY:ROTATE function and then is asked for the degree of rotation and the number of copies. The tablet-menu method has much of the speed of command entry, yet does not force the user to memorize command syntax or to wade through menu levels as in the menu-hierarchy method.

Prompts

"Prompts" are a process in which the system asks the user for information. Preferably, each prompt should be complete and somewhat tutorial, yet concise. For example, "Enter name of drawing to be deleted" is better than "Name." In contrast, "Please enter the name of the stored drawing that you want to be deleted from the flexible disk drive" is too wordy. Prompts of an appropriate length give the user enough information to make an intelligent response.

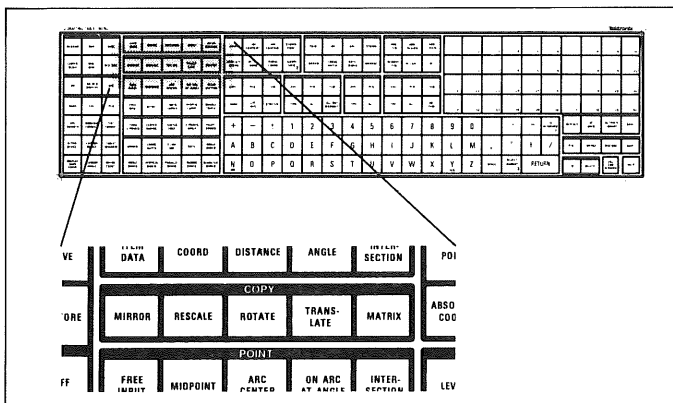


Figure 2. Tablet menus offer much of the speed of command entry without forcing the user to memorize names of parameters.

Because desktop computers are fast they have the advantage here. On a host/terminal system, the slow communication speed (low baud rate) can limit interaction speed with anything beyond very abbreviated prompts.

Menus and yes/no prompts allow the user to see all his options and to select the one that matches what he wants to do. This allows the user to work without having to memorize the options or to repeatedly refer to a manual. For example, if the drafting system requires the user to specify the units for dimensioning, it should show the choices:

Select dimension units

1. Unitless decimal
2. Decimal foot
3. Decimal inch
4. Foot, inch, fraction
5. Inch, fraction

The user should not have to look up dimension choices in a manual or guess and hope that unit type "4" is what he needs (or that "FT/IN/FRAC" is a valid input).

Most prompts should have a default. (A default is what the system assumes if the user does not enter the requested information.) If a default exists, it should be displayed with the prompt. Good default selections serve two important purposes: (1) to help the experienced user work faster, and (2) to suggest a response for a new user who may not understand the full implications of the question. The choice of defaults should follow the principle of "least astonishment," which is discussed in the section on preventing unpleasant surprises.

As much as is possible, prompts and messages should be in the user's language. Although some computer terminology is necessary, the system should avoid computer jargon whenever possible. Most users who would scratch their heads at "I/O device #2 byte count exceeds current capacity" would readily understand "Drawing will not fit on the drawing disk."

A side note: Prompts in both upper and lower case are much easier to read than those in all upper-case. The "shape" of the words can be perceived and understood more quickly.

Input to the System

Desktop computers often have great flexibility on how the user can answer a prompt. (Input to a system is the user's answer to a prompt.) For example, it is easy for the program to receive one character at a time (single-key) and give instant feedback. (On a large computer system, instant feedback is often not practical due to low communication speeds and time-sharing lags.)

A user interface can take advantage of single-key input in several ways. First of all, invalid keys can be screened out immediately by having a friendly bell ring right after the user presses one. Without single-key input, the user would type in a whole line of text and press RETURN only to get a "syntax error" message. He would then have to re-enter the correct text. Single-key input also speeds information entry, since one-character entries like Y and N for yes and no responses require only one keystroke.

When prompted for information, the user should always be provided with a method to quickly "escape" to some known place. For example, pressing the ESC key on the Tek 2-D drafting system immediately returns the user to the point where he selects a function from the tablet menu. Alternatively, pressing RUBOUT returns the user to the previous question in a prompt sequence. Escapes make it very easy to correct mistakes and give the user more control over the system.

Numeric input is the input of parameter values to the system. A system should be able to receive numeric input (and produce output) in the measurement unit selected by the user. For example, suppose a drawing has a scale of 1/4 inch = 1 foot. The user should be able to specify five feet, three and one-eighth inches with something like 5' 3-1/8 instead of 5.2604166. A system that does not provide automatic scaling may even require the user to compute the scaling by hand ($5.2604166 \times 0.25 \div 12 = 0.109592$ for the above distance). When entering numbers, it is much easier if the user can choose from a variety of ways. For example, 3/4, 2-7/8, 35° 16', and 6.02E + 23 all make sense as numbers and should be accepted.

Graphic input is how the user indicates positions on the drawing. Flexibility is extremely important here. The user may want to simply indicate a position with a screen cursor. Usually, he will want to have this indication "snap" to the nearest grid point (like graph paper), but sometimes he may want to ignore the grid. On the other hand, the user might want to "connect" to something already in the drawing, such as the endpoint of a line or arc. Alternatively, he might want to use numbers to indicate the location, either by absolute coordinates or by something relative to a previous entry. Since there is no way the system can predict which method the user wants to use, how should it prompt him? Tek 2-D Drafting addresses this problem with a mode called "free input."

In the free input mode, any time the system needs a graphic position input, it gives the user instant access to any one of nine methods of entry (see figure 3). Initially, in free input, a full-screen cross-hair cursor appears; then the user selects a positioning method by pressing a key. (Valid keys and their meanings are displayed on the screen as shown in figure 4.) Basically, the user can position by snapping to the nearest grid location (or ignore the grid); or he can snap to one of four item types (points, line endpoints, arc endpoints, and symbol connection points); or the user can key in coordinates (either absolute or relative).

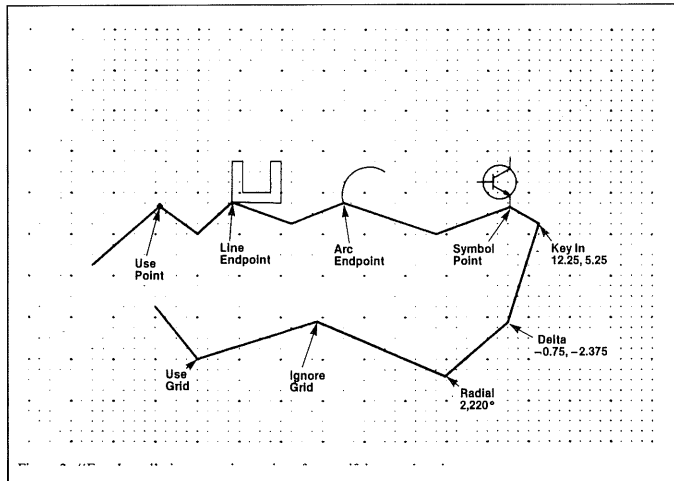


Figure 3. "Free Input" gives the user nine options for specifying any location.

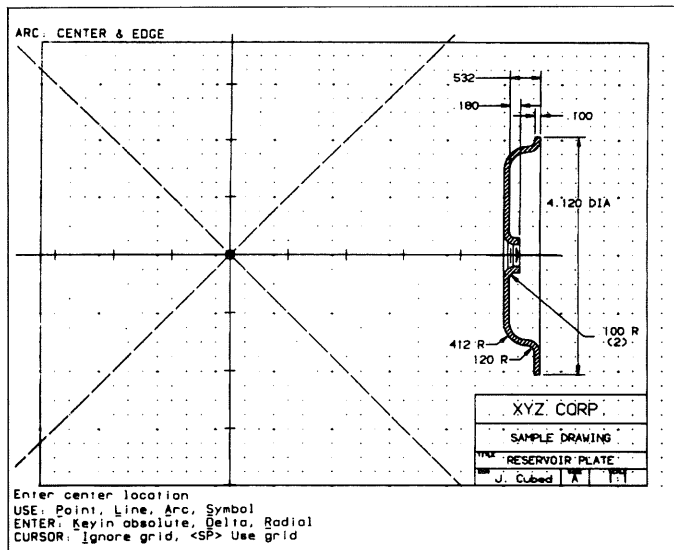


Figure 4. Valid keyboard keys for "Free Input" are displayed (underlined) at the bottom of the screen. This screen hardcopy also shows the cross-hair cursor that is controlled by a set of thumbwheels next to the keyboard.

The flexibility of numeric and graphic input is further extended by allowing the user to enter with either the tablet or the keyboard. He might want to use the tablet to "digitize" or to trace a drawing manually. On the other hand, to create a drawing from scratch, the keyboard thumbwheels might be faster. The user has complete control over the input source, and can even briefly switch from one source to another for just one entry.

Providing Capabilities for Speed and Efficiency

No matter how fast a computer is, there will be times when a user must wait for some operation to finish. The Tek 2-D Drafting system improves user interactions with the system by providing capabilities that minimize this waiting. These capabilities include item selection, redraw, and drawing simplification.

Item Selection and Redraw

Often, in drafting, the user will need to select an item from a drawing. Suppose the user wants to delete a circle; how would he go about it? Typically with drafting systems, you point to the circle on the screen using some sort of cross-hair cursor (like a gun sight). The system then looks for the item closest to the cursor. The Tek 2-D Drafting system uses a special cursor that has a small circle at the intersection of the cross-hair lines. The circle at the cross-hair intersection indicates a search tolerance. If you put part of the item you are selecting inside the circle, the system stops looking as soon as that item is found. With this method, the item will usually be found much faster since it is not necessary to search the whole drawing for the closest item. The Tek 4100 Series has this process implemented in firmware, which makes it even faster.

Anything that reduces the number of items through which the system must search speeds the selection process. The user should be able to select by specifying the type of item. If he is selecting an arc, the user should be able to tell the system to look for arcs only and, thus, by ignoring lines, notes, symbols, dimensions, etc., the system can speed through the selection process.

The order of search also significantly affects selection speed. The Tek 2-D Drafting system follows the "last in, most active" philosophy. In other words, the items most recently added to a drawing are the most likely to be selected for some further operation. For example, it is more probable that the user will want to edit a note just entered than one which was entered yesterday. For this reason, searching is done backwards, that is, from the end of the drawing to the beginning.

Redraw is another operation that benefits from the "last in, most active" philosophy. The Tek 2-D Drafting system provides an option to stop a redraw. Using this option, the user can start the redraw and probably see what was wanted sooner (that is, the items most recently added). He can then stop the redraw and continue working.

Simplifying the drawing

Anything that temporarily simplifies a drawing will speed up virtually all operations. One design approach is to allow the user to turn off certain item types; for example, by telling the system not to display annotation. The disadvantage of this method is that no annotation will be visible and the user may need to see some of it.

A better approach is full *blank* and *unblank* capability. This allows the user to blank (make invisible to both the user and the system) by item type, level, pen, or other criteria. Blanking is even more useful if the user can apply such criteria by region. For example, the user could blank all dimensions that are outside of an area (a box which he indicates on the screen). Blanked items are ignored by the selection and redraw processes until the user unblanks them. Figures 5 and 6 show an example use of blanking.

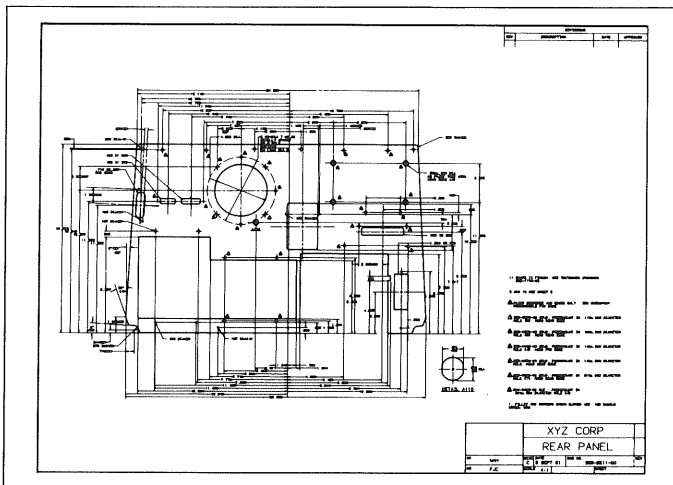


Figure 5. Drawing before blanking.

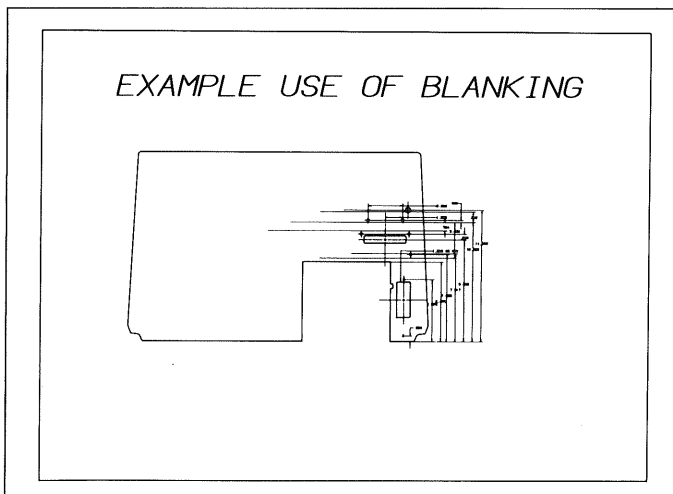


Figure 6. Same drawing as in figure 5 after blanking.

Preventing Unpleasant Surprises and Uncertainty

The design of a user interface should follow what James Foley calls the principle of "least astonishment" (see bibliography). This means that what really happens when the user tells the system to do something should cause the least surprise and shock. In addition, the user should never be left wondering "What's going on – is everything working OK?" Proper implementation of the user interface (and of the system in general) will help prevent unpleasant surprises and uncertainty. This increases the user's confidence in the system and in its ability to do what he wants.

Reliability

Paul Heckel, one of the innovators of the Craig Language Translator, gives a good example of what system reliability really means: Suppose there were two systems. One would get your work done in two hours, and the other would get it done in one hour. Which one would you use? Obviously, the one-hour system. Now suppose you found out that the one-hour system really gets your work done in half an hour, but at the end of the half hour destroys all your work and makes you start from scratch. Which one would you use now?

It is never fun to lose your work. Making a system reliable involves several things.

First of all, the entire system should be "solid." The hardware should work together and should have a reasonable mean time between failures. The software should be virtually bug free. The user should never be hit with a computer error message like UNDEFINED VARIABLE. Even in the event of error, the user should not be left dead in the water. Except for the most extreme hardware failures, users should be able to recover and resume work without losing their drawings.

An additional reliability factor to consider is the ability of a system to recover from a power failure. Recovery is especially important in offices, where users rarely have the protection of an uninterruptable power supply (a special device that maintains equipment power in the event of a brownout or a total power failure).

Another advantage of desktop computer systems is that if one workstation goes down, the others stay up. In comparison, if a central computer goes down, all of the workstations are dead.

Preventing and recovering from user errors

One important way that a CAD system can help a user prevent errors is to make sure the user knows what items on which the system intends to operate. For example, suppose the user wants to delete a line. Once the line is selected from the screen, the line should be highlighted to verify that the correct item was chosen. This could be done by making the particular item blink or change color. The more visible and conspicuous the highlighting is, the less likely it is that the user will make an irreversible error.

The system should ask the user to confirm major actions. In the above example, once the line to be deleted is highlighted, a "Delete item?" yes/no prompt will prevent much distress if the user intended to delete a different line. Potentially destructive operations also can be flagged with blinking warning messages. For example, if the user meant to blank an item but mistakenly selected the delete function, a message that blinks "Warning: deleted items cannot be restored" will help him catch the mistake before it is too late.

Another form of confirmation can be useful when the user is manipulating items. For example, when the user is duplicating an item in the Tek 2-D Drafting system, the first duplication is shown in refresh until the user confirms that it is correct. Only then does the system actually change the drawing.

Another useful function is some sort of a delete-last-item or undo command. This capability allows the user to quickly reverse an action. How much you can undo varies from system to system. Some will only remove the last item entered in the drawing, whereas some systems can actually "unmodify" items that were changed somehow. Once the system starts working on a command, the user should be able to stop it at any time by pressing a special-function key or by some other action. Whenever possible, this interruption should leave things unchanged. For example, if the user cancels an operation that involves overwriting a drawing that already exists on disk, it is best if the system can leave the original on the disk untouched.

There will be times when a user will ask the system to do something impossible, like finding the intersection of two parallel lines. The result should be an error message, and possibly a warning bell so the user will not have to constantly monitor the message areas. (The bell should be friendly, not offensive or irritating.) The error message itself should respect the user's dignity and not "shout." In the parallel-line example, ILLEGAL INTERSECTION!! is too emotional (am I going to jail?). A much better message would be: "Lines are parallel – no intersection possible." Friendly error messages are less traumatic and increase user satisfaction with a system.

Feedback to the user

We have all encountered the uncertainty that occurs when we press a doorbell and hear no ring. Is it working? Should I knock? Is no one home? Am I making a fool of myself standing here? This is analogous for what a user feels when a CAD system does not give instant feedback. We usually don't mind waiting at the door if we know that we were heard and that someone is coming. Likewise, it is much easier to wait for a computer when you know it understood and is performing your command. Timeshared host-based systems, although intrinsically faster than desktop computers, often make the user "wait at the door" without any feedback.

The user interface of a CAD system must provide fast feedback; this can take several forms. When the system user "presses the doorbell," something should happen. If you can hear a "ring," at least you know the system works. Likewise, when the user responds to a prompt on a computer, something should happen instantly. For example, the prompts could disappear.

After you ring a doorbell, it is reassuring to hear responding footsteps indicating that you were heard. Likewise, the computer should indicate that operations are in progress. To do this, the Tek 2-D Drafting system displays a special message – "Working" – any time the computer is busy. The user finds this message is especially reassuring in operations that do not cause display activity, such as saving a drawing on disk. In addition, many commands produce running status messages like "15 items deleted." Some operations even display countdowns that indicate how soon they will be finished.

Feedback is one of the best ways to reduce user uncertainty about a computer system. The more certain the user is, the more comfortable and productive he will be.

Conclusion

User interfaces have come a long way in recent years. The emphasis is moving more and more toward making the user as comfortable with the system as possible.

Desktop computers have helped to speed this change by providing interactive features at very attractive price-to-performance ratios. Some desktop-computer features are unavailable on all but the most expensive host-based systems.

No matter how good the computer hardware is, however, the system is less useful if the computer software provides a user interface that is difficult or unpleasant to use. It is the combination of reliable, highly interactive hardware with reliable, highly interactive software that makes a good user interface. This is what makes a system really friendly, easy to learn, and efficient to use. These factors, in turn, will greatly influence how useful the total system is to the user.

For More Information

For more information, call John Harms, ext. W1-3439. □

This article was developed from material presented at the Design Engineering Conference held in March, 1983.

Bibliography

Foley, James D., Human Factors of User-Computer Interfaces, distributed by Computer Graphics Consultants, Inc., Washington, D.C., 1981.

Foley, James D. and Van Dam, Andries, Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Co., Reading, MA, 1982.

Grimes, Jack D. and Ramsey, Rudy H., "Psychology for User-Computer Interfaces," SIGGRAPH 82 tutorial.

Heckel, Paul, The Elements of Friendly Software, distributed by QuickView Systems, Los Altos, CA, 1982.

Nickerson, R.S., "On Conversational Interaction With Computers," User Oriented Design of Interactive Graphics Systems, (ed. Siegfried Treu), Association for Computing Machinery, New York, 1977.

Savage, Ricky E., Habinek, James K., and Barnhart, Thomas W., "The Design, Simulation, and Evaluation of a Menu Driven User Interface," in Proceedings of Human Factors in Computer Systems, 1982.

Treu, Siegfried, "A Framework of Characteristics Applicable to Graphical User-Computer Interaction," User Oriented Design of Interactive Graphics Systems, (ed. Siegfried Treu), Association for Computing Machinery, New York, 1977. □

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.

COMPANY CONFIDENTIAL
NOT AVAILABLE TO FIELD OFFICES

TECHNOLOGY REPORT
RICHARD E CORNWELL
19-071

DO NOT FORWARD

Tektronix, Inc. is an equal opportunity employer