A DISTRIBUTED NETWORK FOR DEVICE-INDEPENDENT COMPUTER GRAPHICS

by Gary L. Koenig and Clifton Harald
NOI Systems

INTRODUCTION

The purpose of this paper is to describe an approach to implementing a distributed network for device-independent computer graphics software on the HP 3000. A distributed network is a system in which the various elements function as independent processes (sometimes on different computers), cooperating to solve a single problem. Device independence is a feature of a graphics software package that allows a single application program to draw to many different display devices. The discussion will emphasize a single graphics package, the DI-3000 software available commercially from Precision Visuals, Inc., but has broad application to other device-independent graphics application software.

The first section of the paper will provide an introduction to the principles of device independence, and will include descriptions of the primary functional components of device-independent graphics systems. A comparison will then be made between distributed and modular software networks, drawing on actual experience with the DI-3000 package. Originally designed for distributed network implementation, DI-3000 has most often been used in modular software networks in which all modules are simply program subroutines. Within this section, the disadvantages of using modular software networks on limited address machines like the HP 3000 will be addressed in detail.

An approach to implementing a true distributed network will then be presented, again using the DI-3000 software as a specific case study. This part of the paper includes publication of the results of testing recently conducted to determine optimal methods of interprocess communication. A description of the benefits of distributed networks will be presented, substantiated by the results of performance testing. The paper concludes with a discussion of the potential for expanding distributed networking to support multiple CPUs and highly intelligent devices.

DEVICE INDEPENDENCE

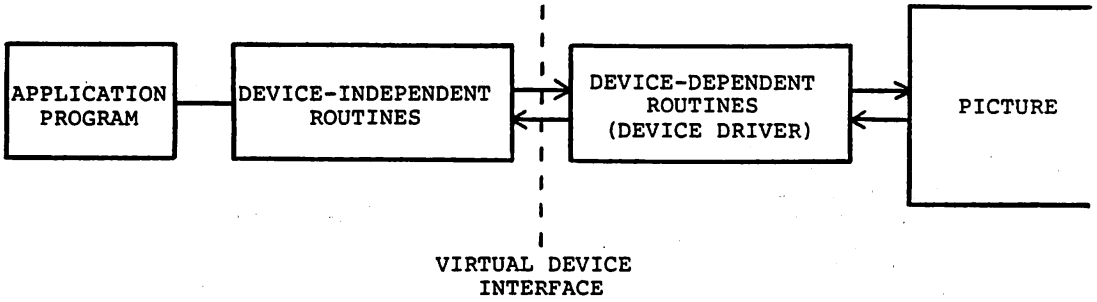
The objective of device independence is to enable a person to display the same or similar graphic images on many different graphics display devices. This capability is desirable for many reasons. Perhaps most importantly, device independence can minimize the effects of hardware obsolescence, since new display devices can replace outdated ones without changing graphics application programs. Secondly, graphics applications can be developed initially on inexpensive equipment and then, through device independence, transferred to more sophisticated devices for detailed modification. Additionally, device-independent software is not limited to a specific vendor's hardware inventory.

In a truly device-independent graphics system, general graphic requirements of a single application program are translated by software modules (called device drivers) into specific graphic commands for a target display device. Commands may be passed to the device as a string of ASCII characters, as binary information or in whatever form is required. Figure I depicts the primary functional components of a device-independent graphics system. An application program calls up device- independent routines which perform basic calculations and format an image in nondevice-specific terms. When the image has been properly formed, it is passed to a device driver through a virtual device interface. The virtual device interface functions as a communication link between the device-independent routines and one or more device drivers. Since each graphics display device has different capabilities and a different command language, separate device drivers are required for each device supported by the

graphics package. The device driver translates the image into device-specific commands which are then sent to the display device, resulting in representation of the image on the device.

FIGURE I

Device-Independent Graphics System



The key to device-independent graphics is the virtual display device, which is a hypothetical device that represents the combined graphics capabilities of all the devices actually supported by a graphics package. All graphics computations are oriented toward the actual device requirements represented by the virtual display device.

Since a virtual display device is an idealized representation of all graphics device capabilities, very few devices fully support the virtual device. Those functions that cannot be performed directly by the device must be either simulated or overlooked by the device driver. Capabilities such as line style, line width, polygon fill, polygon interior patterns, text attributes, and image transformations generally can be simulated. Color, intensity, real-time motion, and input capabilties generally must be overlooked if not specifically supported by the device hardware.

Device independence has been implemented in a number of different ways over the past decade. Current standardization efforts are attempting to provide a common basis for designing device-independent packages. One of the earliest set of standards was the Core package forwarded by the Graphics Standards Planning Committee of ACM/SIGGRAPH. The Core standard has recently given way to the international Graphical Kernal System (GKS) standard, which has been adopted by the ANSI X3H3 committee.

THE CORE STANDARD

Although the Core standard has been superseded by the GKS standard, it is still noteworthy, because a number of highly successful graphics packages are based on it. The Core system defines standards for the following graphics package features:

- o System and virtual device control;
- o Positioning and nontext primitives such as moves, lines, polylines, polygons, and markers in either two- or three-dimensional coordinate systems;
- o Attributes for positioning and nontext primitives. These attributes include color, intensity, linestyle, line width, polygon

edge style, polygon interior style and marker symbol;

- Text, text primitives and text attributes such as path, font, justification, size, gap, and base;
- Segments and segment attributes. A segment is a collection of output primitives making up a part or all of a graphic image. Segment attributes include visibility, highlighting and pickability, i.e., whether the segment can be uniquely selected from an interactive graphics device. Segments can be defined temporarily, or retained permanently;
- Transformations. Modeling transformations allow an object to be translated, scaled or rotated within its own coordinate system; Viewing transformations define the position orientation, line of sight and lens configuration that describe how the image will look when displayed on the graphics display device. Image transformations allow the displayed images to be translated, rotated or scaled on the display device surface;
- Virtual graphics input;
- Inquiry capabilities allowing the graphics system to determine the current state of all attributes and primitives.

One of the most intriguing challenges of implementing a Core-based system is to determine where to place the virtual device interface. Placement is typically defined in terms of its proximity to the application program. Figure II depicts a simple virtual device interface that is far from the application program. Figure III shows the interface closer to the program. As these illustrations suggest, the major complexity of a graphics software package can be found in either the device independent segment or the device driver segment of the system, depending on the placement of the virtual device interface.

FIGURE II

Simple Virtual Device Interface

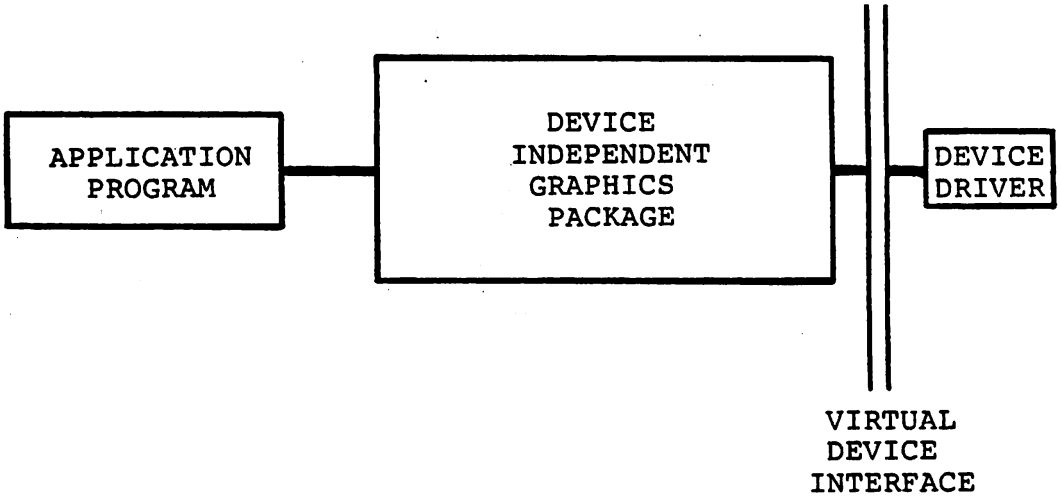
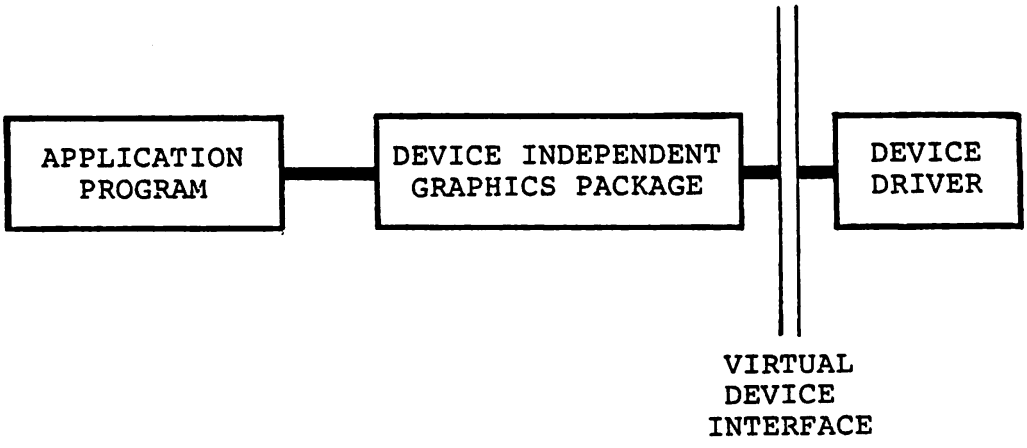


FIGURE III

Intelligent Virtual Device Interface



As the sophistication of graphics devices improves, it is generally felt that the virtual device interface should be moved closer to the application program, in order to take full advantage of the device features available. If the device has limited capabilities, the device driver will either simulate or override requests for functions not provided in the hardware.

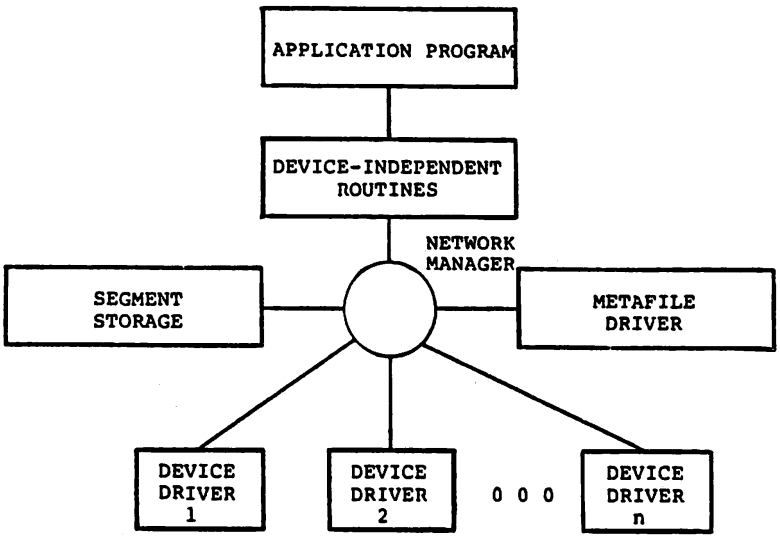
THE DI-3000 SOFTWARE NETWORK

DI-3000, developed by Precision Visuals, Inc. of Boulder, Colorado, is a commercially available system based on the 1979 Core standard. Designed as a complete three-dimensional coordinate system, treating two-dimensionality as a subset, DI-3000 is available in two configurations: Level A and Extended. Level A offers all the Core system features except for retained segment features. The Extended configuration adds retained segment capabilities to the Level A configuration.

DI-3000 was designed as a modular graphics software network. This approach lends itself to distributing tasks to intelligent graphics devices or among linked Central Processing Units. In this system the virtual device interface is located close to the application program, allowing DI-3000 to take full advantage of device capabilities. Figure IV illustrates the distributed network of DI-3000.

FIGURE IV

DI-3000 Distributed Network



The DI-3000 network is composed of four types of modules. In the first module, the application software is combined with the device independent routines. Device drivers reside in separate modules. Theoretically, multiple devices can be activated concurrently, and all devices can be selected at run time. The metafile driver module shown above is similar to a device driver, but "draws" to a disk file rather than to a display driver. These metafile images can be redrawn and/or modified at a later time by using a special metafile translator program.

All retained segments are stored in, and managed by, the segment storage module. Segment storage also provides support to the device drivers. Intelligent devices that maintain display lists (or segment information), do not need much support from the segment

storage mode. A command to make a segment invisible would be handled by such a device. A less intelligent device, however, would have to depend on the device driver and segment storage to erase, redraw or otherwise perform the software functions necessary to simulate segment invisibility.

As shown in Figure IV, the network manager is the hub of the DI-3000 network; all communication between nodes must go through the network manager. A sophisticated protocol has been developed to allow efficient message switching among the nodes. All messages sent through the network manager are made up of fifteen-bit, unsigned integers, or two eight-bit ASCII characters. In either case, the basic transmission unit is a sixteen-bit "word." The obvious advantage of this method is the ease of implementation on any computer that has at least a sixteen-bit word size. All of the modules in the network are tailored to the word size of the host CPU. The sixteen-bit restriction is in effect only for messages routed through the network manager. This approach enables a sixty-bit machine to host the application program and device independent routines while other nodes reside on a sixteen-bit CPU. The major disadvantage of the fifteen-bit resolution limitation is that it might be insufficient to drive ultra-high resolution devices.

COMMON IMPLEMENTATION OF DI-3000

Although designed as a distributed network, DI-3000 is most commonly installed as a subroutine library system, where all necessary routines (including the device driver), are bound together in a single program image. Although this approach is satisfactory for many applications, it does not take full advantage of the distributed features of the network. For example, device selection cannot be deferred until run time, because a device must be bound to the program at PREP time. Furthermore, since all routines are bound into a single program image, it is difficult to support multiple devices concurrently. Though typically used on the HP 3000, single bound program implementation presents difficulties due to HP architectural limitations.

The major concern in implementing DI-3000 on the HP 3000 arises from the product's memory requirements and the machine's memory management limitations. Figure V presents the code segment and data segment requirements for the DI-3000 routines. After these requirements are met, little data segment remains for the application program. As a result, the usefulness of a toolset product like the DI-3000 is restricted.

FIGURE V

DI-3000 Segment Requirements

	Approximate Data Segment Used	Approximate Code Segment Used
DI-3000 with HP7220 Device Driver	40K bytes	70K bytes
DI-3000 Extended with HP7220 Device Driver	47K bytes	102K bytes

The amount of code segment used raises special concerns. As noted above, DI-3000 is a library of callable subroutines, but the HP 3000 does not have well- developed library handling tools. Therefore, nearly all routines are bound to the application program, even if they are not required.

Neither RL or SL files provide an adequate solution to the code segment usage problem. Most DI-3000 routines rely on the use of COMMON (global) storage. As a result, they cannot be placed in an SL file. Although an RL file could be used for some routines, it would not serve them all, because routines loaded from an RL file are placed into the same code segment. As Figure V shows, the amount of code segment memory required would normally exceed the system limitation for a single code segment. If HP would allow multiple RL searches, however, the RL alternative would offer great promise.

Due to these limitations, most DI-3000 routines are placed into a single USL and the program is PREPed using the USL. PREP time is often longer than necessary, because many routines not required by the application program must be linked into the program image.

Since SL use is prohibited, very little code sharing is possible. Even though the code for the DI-3000 routines is identical for two different application programs, code sharing is impossible when the two programs are run simultaneously. Only when users are running the same program files can the advantages of code sharing be realized.

The Extended version of DI-3000 also presents limitations in usefulness. Since there is little data segment available for the application program, it is difficult to dedicate a significant amount of memory space to the segment storage module. An application program that has complex segment storage requirements is virtually impossible to create on the HP 3000 using bound program image implementation.

PROCESS HANDLING IMPLEMENTATION OF DI-3000

Since DI-3000 was designed for a distributed network environment, its implementation on the HP 3000 may be facilitated by using MPE's process handling capabilities. Implementation using process handling is straightforward due to the rigid separation between modules in the DI-3000 network. Since all communication between modules goes through the network manager, the device drivers, metafile driver and segment storage module may be regarded as "son" processes, invoked and controlled by the network manager. The network manager may then be bound together with the device independent routines and application program into a single program image.

Once the decision to utilize process handling is made, it is necessary to determine which interprocess communication alternative to use: message files, extra data segments or the RECEIVEMAIL and SENDMAIL intrinsics. The choice is not obvious, however, because the network is not being implemented for simultaneous processing purposes. Message files are the most reasonable alternative when the processes operate simultaneously, and must communicate with each other at irregular intervals. In DI-3000, however, the network manager passes a message to a device driver (or other "son" process), and then waits until a reply is sent back before attempting to perform any other function.

INTERPROCESS COMMUNICATION TESTS

To test the resources used by each of the communication methods described in the preceding section, several relatively simple FORTRAN test programs were designed. The "father" processes were coded as follows:

Message File	Extra Data Segment	MAIL
Find the time of day	Find the time of day	Find the time of day
Arbitrarily initialize a 256-word buffer	Arbitrarily initialize a 256-word buffer	Arbitrarily initialize a 256-word buffer
Build and FOPEN an input Message File	Get a 256-word Extra Data Segment	
Build and FOPEN an output Message File		
Create and activate the "Son" process	Create the "Son" process	Create the "Son" process

Message File	Extra Data Segment	MAIL
Send and receive 1000 256-word buffers using FWRITE and FREAD	Send and receive 1000 256-word buffers as follows: DMOVOUT the buffer; activate the "Son" process; suspend itself; when activated by the "Son,"	Send and receive 1000 256-word buffers as follows: SENDMAIL the buffers; activate the "Son"

	DMOVIN the buffer	process; suspend itself; when activitated by the "Son," RECEIVEMAIL the buffer
Find and display the "Father's" CPU time	Find and display the "Father's" CPU time	Find and display the "Father's" CPU time
Find and display the Elapsed time	Find and display the Elapsed time	Find and display the Elapsed time
Exit	Exit	Exit

In all cases, the "son" processes essentially performed mirror actions. The programs were run on a Series 40 with 1 megabyte of memory and two Winchester drives on a single GIC. One set of tests was run on a stand-alone machine, and another on a machine that had reached steady-state conditions under a heavy, reproducible load. The latter tests were designed to heavily stress the CPU, memory and disc channel.

Figure VI presents the results of running each test five times under no-load and heavy load conditions. The tests consistently demonstrated the advantage of using the extra data segment method. Although the coding of this method is slightly more difficult than the coding of the message file method, the inconvenience is compensated for by lower execution and elapsed times. As a result of these tests, DI-3000 was implemented as a process handled network, passing messages through a 256-word extra data segment.

FIGURE VI

Interprocess Communication Test Results
(All times in seconds)

NO LOAD

	Message File			Extra Data Segment			MAIL		
Test	"Son" CPU	"Father" CPU	Elapsed Time	"Son" CPU	"Father" CPU	Elapsed Time	"Son" CPU	"Father" CPU	Elapsed Time
1	1.50	1.76	8.19	0.71	0.84	5.07	1.40	1.50	16.38
2	1.47	1.85	8.26	0.71	0.84	4.88	1.43	1.54	16.44
3	1.46	1.85	8.27	0.72	0.84	4.84	1.44	1.54	16.39
4	1.47	1.85	8.21	0.72	0.84	4.81	1.44	1.53	16.39
5	1.45	1.86	8.18	0.71	0.84	4.82	1.42	1.54	16.41
Average	1.47	1.83	8.22	0.71	0.84	4.88	1.43	1.53	16.40

WITH LOAD

1	1.43	1.76	10.28	0.71	0.82	7.49	1.50	1.62	19.71
2	1.44	1.79	10.47	0.71	0.82	7.49	1.60	1.68	19.64
3	1.44	1.78	10.60	0.69	0.82	7.32	1.58	1.68	20.92
4	1.42	1.79	10.49	0.70	0.81	7.37	1.58	1.68	20.50
5	1.43	1.79	10.36	0.71	0.81	7.36	1.59	1.70	21.55
Average	1.43	1.78	10.44	0.70	0.82	7.41	1.57	1.67	20.46

BENEFITS OF THE PROCESS STRUCTURE

By using HP's process handling capabilities, the advantages envisioned in the design of the network have been realized. Furthermore, most of the difficulties arising from HP hardware and software limitations have been overcome. Through process handling, for example, graphic devices can be selected at run time. Device drivers are not bound to the program image, since they run as "son" processes. A special routine was added to DI-3000 to allow the programmer (and ultimately, the user) to decide which device driver to activate at any time during program execution.

Using process handling, drawing to more than one device at the same time presents no special problems. Although DI-3000 was designed to support multiple devices concurrently, single bound program image implementation practically precludes use of this feature. This is because many of the same routine and common block names are used in all device drivers. Since no device drivers are bound in the process handling implementation, users of the network are not limited to a single device driver.

It is important to remember that a "son" process must always respond to the network manager before further processing takes place in the device independent part of the network. This allows use of the same extra data segment to communicate with all "son" processes. Consider, for example, a program that draws a pie chart to an HP2623 terminal and an HP7221 plotter simultaneously. The device independent routines form a slice of the pie in a virtual device form. The network manager activates the HP2623 device driver and sends a message through the extra data segment. The HP2623 device driver translates the message, draws the slice on the screen, and then replies to the network manager. The network manager then activates the HP7221 driver and sends the same message through the same extra data segment. The HP7221 driver then translates the message, draws the slice and replies to the network manager. This same sequence is repeated until the entire pie chart has been drawn on both devices.

Another benefit of process handling is a large increase in the amount of data segment available to the application program. This is a result of all global and local storage in the device drivers, metafile driver and segment storage being moved out of the "father" data segment and into data segments for the "son" processes. DI-3000 now takes approximately 20,000 bytes of data stack for both Level A and Extended versions. This requirement is a substantial improvement over the 40-47,000 bytes required when everthing is bound into a single program image. As a result of increased data segment availability, the programmer not only has more data stack, but considerably more segment storage, as well. Segment storage is a separate process with its own data segment, which allows programmers to dedicate more than 25,000 words to segment storage.

Under process handling implementation, significant improvements in the efficiency of program creation are realized. The time required to PREP an application program is diminished, as is the complexity of the operation. USL management is more efficient, because the USL does not contain code from device drivers, segment storage and the metafile driver. Similarly, the Segmenter takes less time to PREPARE a program because it has fewer routines to link. The only detrimental aspect of process handling is an increase in drawing time. Although the increase has not been measured and will vary with the baud rate and display device, it is expected to be no more than ten percent in the worst case.

One of the most important benefits of process handling is that all "son" processes may be shared, since the program image of these processes does not change from application program to application program. Sharing of "son" processes thus relieves some of the strain that DI-3000 can place on memory.

A potential benefit that has not yet been fully implemented is the ability to call DI-3000 routines from COBOL, BASIC or PASCAL. Almost all FORTRAN input and output takes place in the device drivers and the metafile driver. The remaining FORTRAN input/output mainly addresses error and debug processing. Once the latter input/output is transferred to a "son" process, or converted to system intrinsics, DI-3000 should be available to host languages other than FORTRAN.

FUTURE POSSIBILITIES

Two extensions of process handling principles offer potential benefits in future applications: concurrent multiple processing, and distributed processing using more than one computer.

Concurrent multiple processing offers the most potential in an environment where many different DI-3000 application programs access a large number of devices. In such an environment, the optimal system configuration would most likely call for separation of the network manager from the device-independent module, placing it in a global process that communicates with several application programs, as well as all device drivers, segment storage and the metafile driver. The application programs would asynchronously send messages to the network manager to be forwarded to the appropriate driver or segment storage.

The network manager would thus direct the flow of signals between all programs and processes.

Distributed processing offers even more potential, given the increasing intelligence of graphics devices. A network in which device drivers actually execute functions within specific devices would lighten the load placed on the CPU and the main memory. It is significant that a device driver that runs and draws on the IBM PC is currently available. True distribution of processing will continue to prosper as more graphics devices gain a high level of intelligence.

SUMMARY

Device independence is a very desirable, perhaps indispensable, feature for any graphics package to possess. Standards have grown up around this feature and have, in turn, been implemented as commercial packages.

DI-3000, the package described in this paper, was designed as a software network. Performance problems arise when DI-3000 is simply installed as a library of subroutines. Taking advantage of the modular design by implementing DI-3000 as a process handled network releases its full power while minimizing its effect on system resources.

The network design can be extended to allow some processes to run on separate computers, especially intelligent graphics devices. This is a practical and efficient example of a true, distributed processing network.

BIBLIOGRAPHY

Olenchuk, Bruce. "Graphics Standards." Computer Graphics World; Volume 6, Number 8. August 1983.

Precision Visuals, Inc. DI-3000 User's Guide. Boulder, CO: Precision Visuals, Inc.

Warner, James R. "Principles of Device-Independent Computer Graphics Software." Los Alamitos, CA: The Institute of Electrical and Electronics Engineers, Inc., 1981.

BIOGRAPHICAL SKETCHES

Gary L. Koenig is a computer consultant for NOI Systems of Boulder, Colorado. He provides a wide range of services for HP 3000 sites, but specializes in computer graphics, accounting and productivity tools. In the past two years, Koenig has supervised development projects utilizing many of HP's development tools: FORTRAN, COBOLII, PASCAL, and VPLUS.

Koenig's past experience includes system programming, DP planning, Data Center management, customer support, product planning, system performance, computer networking, and application programming.

Koenig received a BS in Mechanical Engineering from Iowa State University in 1969. He worked briefly as an application engineer before joining the Data Processing profession.

Clifton Harald is a Research Associate with the National Institute for Socioeconomic Research (NISR) in Boulder, Colorado. He specializes in evaluating the public cost and revenue consequences of community development trends, and in

analyzing alternative local government fiscal management policies. He has managed several socioeconomic impact assessment projects for NISR, and directs marketing of NISR's Planning and Management Services among local governments in the western United States.

Harald also has experience in energy and environmental planning, having previously worked as an Energy Specialist with the cities of Seattle, Washington, and Boulder, Colorado. As an Environmental Planner with the regional transit agency in Seattle, he coordinated facility planning for multi-million dollar construction projects and prepared technical analyses of transit system energy use.

A graduate of the University of Washington, Harald holds a Masters degree in Urban Planning. He received his Bachelor of Arts degree in Literature, Sociology and Psychology from the University of Colorado.
