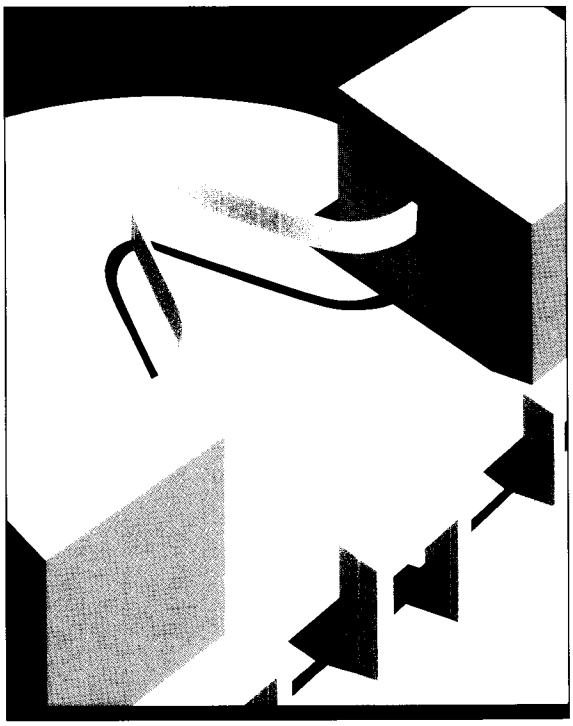
T A N D E M

SYSTEMS REVIEW

VOLUME 2, NUMBER 2 JUNE 1986

BRANDIFINO



Distributed System Protection

PATHWAY IDS ▪ New TAL Features

Predicting Response Time

DP1-TMF Cache Sizing

New Disk Technology

Margin Analysis

New Products ▪ Technical Paper

Corrections:
Please note the following corrections to the article entitled "DP2 Performance," which appeared in the June 1985 issue.

Page 37, Figure 4. The caption should read:

> The elapsed time required for A06 DP1 and B00 DP2 to copy 1,000 100-byte records sequentially from one file to another on the same disk.

Page 38, Table 1. The title should read:

> The elapsed time required by A06 DP1 and B00 DP2 to copy 1,000 100-byte records sequentially from one file to another on the same disk.

Page 41, Table 3. All digits in this table are for Sends and Receives both for messages and bytes per message. For an accurate picture of message traffic divide the numbers by 2.

Volume 2, Number 2, June 1986

Editor
Ellen Marielle-Tréhoüart

Technical Advisor
Dick Thomas

Associate Editors
Wendy Osborn
Carolyn Turnbull White

Assistant Editor
Sarah Rood

Art Director/Cover Art
Stephen Stavast

Production and Layout
Phil Just
Steve Kirtley
Stephen Stavast
David Thompson
John Tomasini

Typesetting
Barbara Cowlishaw

The *Tandem Systems Review* is published by Tandem Computers Incorporated.

Purpose: The *Tandem Systems Review* publishes technical information about Tandem software releases and products. Its purpose is to help programmer-analysts who use our computer systems to plan for, install, use, and tune Tandem products.

Subscription additions and changes: Subscriptions are free. To add names or make corrections to the distribution data base, requests within the U.S. should be sent to Tandem Computers Incorporated, Sales Administration, 19191 Vallco Parkway, MS 4-05, Cupertino, CA 95014. *Requests outside the U.S. should be sent to the local Tandem sales office.*

Comments: The editor welcomes suggestions for content and format. Please send them to the *Tandem Systems Review*, 1309 So. Mary Avenue, Sunnyvale, CA 94087.

Distributed System Protection with SAFEGUARD

As hardware has become less expensive and business demands for data processing have grown, systems have been developed that allow the user to distribute applications and data. While this capability may make a business easier to run, it makes the job of protecting information that much harder.

In a single-processor system, the user could simply protect the resources physically; with distributed processing, that is no longer adequate.

In a distributed system, data must be protected wherever it resides in the network, as well as while that data is being transported between nodes in the network. Therefore, a distributed system protection mechanism protects data while it is on-system (e.g., in the processor, on a disk drive) as well as when it is off-system (e.g., traveling through communication lines, on removable disk packs) as shown in Figure 1.

This article explains the on-system protection mechanisms provided in the Tandem™ NonStop™ system and describes the relationship between each of these mechanisms. While the protection features of the GUARDIAN 90™ operating system are equivalent to those found in many other commercial mainframe operating systems, a distributed system architecture poses some special challenges.[1] The SAFEGUARD™ family of products was designed to extend the protection features of the GUARDIAN 90 operating system.

Every computer system is composed of three fundamental layers: the hardware, the operating system, and the application. Protection mechanisms at each layer of the system are built on, and depend upon, the correct implementation of the mechanism provided at the layer below it.

Security of information is achieved only through a balanced application of policies that are enforced by protection mechanisms. This highlights a very important principle: the separation of policy and mechanism. Mechanisms determine how to do something. Policies decide what will be done, but not how it will be done. Corporate policies can and do change to reflect changes in economics as well as the law. Without the effective separation of mechanism and policy, the ability of corporate computer systems to adapt is severely impaired.

[1]For more information on the off-system protection mechanism, see the *Data Encryption Standard*, 1977, and Ehrsam, et al., Vol. 17, No. 2.

Hardware Protection

In a computer system the needs and privileges of users and applications vary, and they differ as a whole from the needs and privileges of the operating system itself. The hardware provides protection features that isolate the executing programs, protect the operating system from user programs, and allow only the operating system to perform such sensitive operations as physical I/O. The hardware provides the ability to protect both operating system instructions and operating system data from the user.

Instruction Protection

Certain machine instructions are intended for use by the operating system only. These are called privileged instructions. The hardware allows privileged instructions to be executed only when the processor is executing in privileged state. In the NonStop II™, TXP™, and EXT™ processors, if an attempt is made to execute a privileged instruction in user mode, the hardware does not execute it, but treats it as an illegal instruction and traps to the operating system. Privileged instructions perform functions such as halting the processor, issuing I/O requests, and making the transition from user mode to privileged mode. Any code operating within the privileged state must be considered part of the protection kernel and, therefore, a trusted piece of code.

To control the execution of privileged operations and to prevent a nonprivileged process from executing in privileged mode, every procedure in the Tandem NonStop system has one of three attributes:

Nonprivileged. Procedures with this attribute can be called by any procedure. They execute in the same mode (privileged or nonprivileged) as the calling procedure. This attribute is typically given to all of the procedures in an application program.

Callable. Procedures with this attribute can also be called by any procedure, but they execute in privileged mode. The caller's mode is restored when a callable procedure exits. Such procedures provide a controlled interface between a nonprivileged application program and the privileged operating system.

Privileged. Privileged procedures execute in privileged mode and are callable only by procedures currently executing in privileged mode. An attempt by a nonprivileged procedure to call a privileged procedure results in control being transferred to the operating system instruction-failure trap handler. Procedures are given this attribute when their functions, if performed improperly, could have an adverse effect on the processor's operation. A nonprivileged application program's only interface to a privileged procedure is through a callable procedure.

This mechanism is implemented in the NonStop II, TXP, and EXT procedure call instructions. If the calling procedure is not executing in privileged mode, the "callability" attribute of the procedure being called is checked. When a procedure is exited, the mode (privileged or nonprivileged) is reestablished to be the lesser of the caller's setting and the current settings. This prevents a nonprivileged caller from being left in privileged mode on return from a callable or privileged procedure.

Figure 1

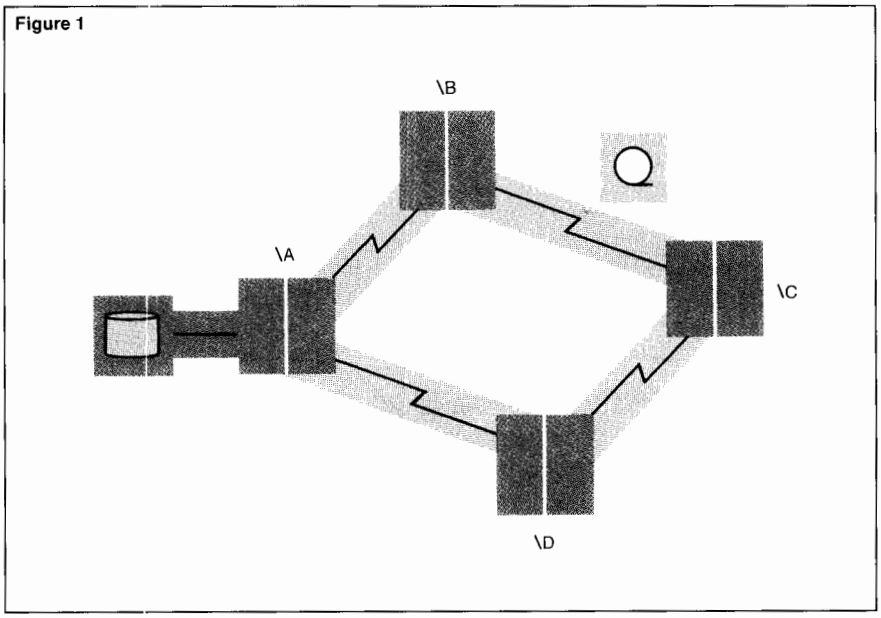
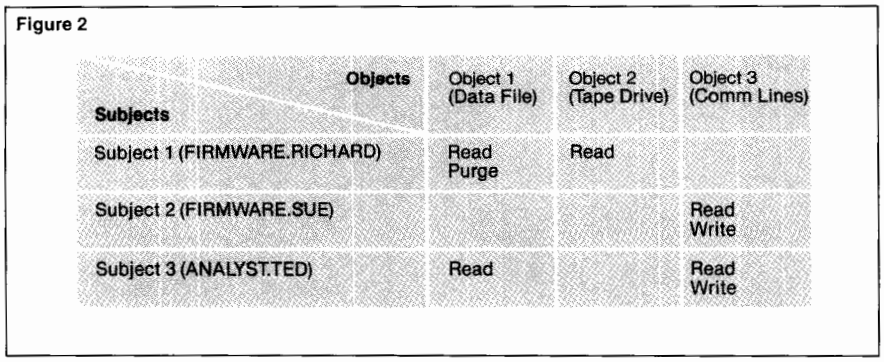


Figure 1.
Integration of SAFEGUARD and SAFE-T-NET™ protects data while it's on or off a Tandem system.

Figure 2



Subjects \ Objects	Object 1 (Data File)	Object 2 (Tape Drive)	Object 3 (Comm Lines)
Subject 1 (FIRMWARE.RICHARD)	Read Purge	Read	
Subject 2 (FIRMWARE.SUE)			Read Write
Subject 3 (ANALYST.TED)	Read		Read Write

Figure 2.
Access matrix.

Data Protection

Although its primary function is to expand the memory available to the programs, virtual memory is a valuable protection feature. The virtual memory mechanism maps a logical view of memory onto the physical storage. The logical memory is memory as the process views it. In general, a process is allowed to see only a subset of the virtual memory, consisting of data and code that it owns. For a nonprivileged process on the NonStop II, TXP, and EXT systems, logical memory is separated into seven address spaces:

- User data.
- System data.
- User code.
- System code.
- User library.
- System library.
- User extended data.

The system data, system code, and system library address space belong exclusively to the operating system. This mechanism controls the user's ability to modify (either accidentally or deliberately) the operating system code or data structures.

Operating System Protection

The operating system uses the hardware protection mechanisms to protect itself from users and their application programs. Relying on this, the operating system protects users from each other.

To clarify, consider a basic protection model. One of the best known protection models is the access matrix model (Lampson, 1974).

The basic elements of the model are subjects, objects, and operations:

- A subject is an active entity capable of accessing objects; e.g., users are subjects.
- An object is anything to which access is controlled, such as data files or tape drives.
- An operation is simply a kind of access to an object, such as read or write access.

For each type of object there is a set of possible operations. Data files, for example, can be accessed through operations such as read or write.

An access matrix relates the three types of elements of the model as shown in Figure 2. In this matrix the rows represent subjects and the columns represent objects. Each cell contains a list of operations permitted to subject i for object j. Access (i,j) defines the set of operations that a process executing as subject i can invoke on object j. When implementing access control, it is generally inefficient to represent the information as a matrix, because the matrix is typically sparse. That is, there are many objects and subjects but there are relatively few operations specified.

Two techniques are commonly used for storing the information:

- An *access control list* associated with the object that lists all of the subjects that can access the object, along with the operation.
- A *capability list* associated with a subject that lists all the subject's rights to objects.

Figure 3 shows the information presented in Figure 2 in an access list, and Figure 4 shows the same information in a capability list. In general, most systems contain both access and capability lists (Saltzer and Shroeder, 1975). The GUARDIAN 90 operating system and its extension, SAFEGUARD system protection software, implement the access matrix predominately as an access control list.

Subjects

A GUARDIAN 90 subject is more commonly called a GUARDIAN 90 user. The GUARDIAN 90 operating system defines both local and network users. For a single system (i.e., a system with all processors connected using a DYNABUS™ architecture), the definition of a local user is not fundamentally different from many other commercial mainframe operating systems. However, since GUARDIAN 90 is also a network operating system we must also have a definition of a network user.

Local User. Each GUARDIAN 90 user has a unique user name and a corresponding unique user ID. A user name is composed of two elements: <groupname>.<username>. The first term, <groupname>, is the name of the group to which the user belongs; <username> is a name identifying the individual user within the group.

Similarly, a user ID is of the form <group id>,<user id>, where <group id> identifies the user's group and <user id> identifies the user within the group. For example, a user named Richard Carson, who works in the firmware group, has a GUARDIAN 90 user name of FIRMWARE.RICHARD and a user ID of 3,104. This would also mean that all other users in the firmware group would belong to <groupname> FIRMWARE and <group id> 3.

Network User. The GUARDIAN 90 operating system is a network operating system, which means that it supports the concept of a network user. A network user is any user who has the same user name and user ID on more than one system in the network and has matching remote passwords between those systems. For example, if a user (USERID: 3,104 USERNAME: FIRMWARE.RICHARD) exists on the system named \TIBET and the same name and ID exist on \RIO, and if the user has matching remote passwords on \TIBET and \RIO, then he is a network user. If he makes a request from a process on \RIO for a resource on \TIBET (e.g., attempts to open \TIBET.$DATA.PAY.ROLL), then \TIBET sees the request as coming from a remote version of FIRMWARE.RICHARD.

Figure 3

Object 1	Subject 1 (Read, Purge), Subject 2 (Read)
Object 2	Subject 1 (Read)
Object 3	Subject 2 (Read, Write), Subject 3 (Read, Write)

Figure 3.
Access control list.

The difference between a local and remote version of a user in GUARDIAN 90 is analogous to the difference between recognizing individuals in person and recognizing them across a telephone line. Accordingly, GUARDIAN 90 protection mechanisms differentiate between these two versions of a user.

One fundamental principle (and policy) that the GUARDIAN 90 operating system implements is that of equal distrust of all remote systems. In other words, from \TIBET's viewpoint, a FIRMWARE.RICHARD authenticated on \BORA and a FIRMWARE.RICHARD authenticated on \RIO are no different. They are both remote versions of the network user, FIRMWARE.RICHARD, and are treated as such.

Objects

At the highest level, the system entities to be protected are referred to as objects. Each GUARDIAN 90 object is identified by a unique file name. A file can be all or a portion of a disk, a device such as a terminal or line printer, or a process. A file is referenced by the unique symbolic file name that is assigned when the file is created.

Figure 4

Subject 1	Object 1 (Read, Purge), Object 2 (Read)
Subject 2	Object 3 (Read, Write)
Subject 3	Object 1 (Read), Object 3 (Read, Write)

Figure 4.
Capability list.

For a disk file the symbolic name has three parts:

- A volume name to identify a particular disk pack in the system.
- A subvolume name to identify the disk file as a member of a related set of files on the volume.
- A disk file name to identify the file within the subvolume.

An example is $DATA.PAY.ROLL.

A device file (e.g., terminal, line printer, magnetic tape unit, card reader, data communication line) is referenced by a symbolic device name or logical device number. $TERM1 is an example of a terminal device. Device names and their corresponding logical device numbers are assigned at system configuration time.

Finally, for a process file there are two mutually exclusive forms of the identifying process: the timestamp form and the process name form. The timestamp form contains (among other things) the time the process was created. The process name form uniquely identifies a process or process pair in the system. An example of a process name is $UPD.

GUARDIAN 90 Protection

The GUARDIAN 90 operating system provides some basic protection mechanisms. When a user logs on to use GUARDIAN 90 interactively, or when a program programmatically logs on as a user, the operating system uses passwords to ensure the authenticity of the user before allowing that process to take on the identity of the user. Once the user has been authenticated, GUARDIAN 90 sets the user ID of the process requesting the LOGON to that of the user. The process then is allowed to gain access to information in the system as an agent for the user.

The main focus of the GUARDIAN 90 operating system is the protection of disk files. Four operations are controlled: read, write, execute, and purge. These accesses are defined as follows:

- *Read*, meaning examine or copy the disk file's contents.
- *Write*, meaning modify the contents of the disk file.
- *Execute*, meaning execute the file (if it is a program file).
- *Purge*, meaning delete the file.

The disk file's owner can establish one of seven levels of protection for each of the four types of access. All of these levels are relative to the owner of the disk file on the given system:

- Local Super ID only.
- Local or network owner only.
- Any member of the local or network owner's group.
- Any network user.
- Local owner only.
- Any member of the owner's local group.
- Any local user.

Some basic protection services are inherent in GUARDIAN 90. There are, however, certain services lacking in the GUARDIAN 90 protection mechanism. SAFEGUARD software was designed to correct and enhance many of these areas:

- Insufficient granularity (the GUARDIAN 90 protection mechanism provides coarse access control to disk files). For example, in order to share a file between two user groups in the system, access to the file must be permitted to all users on that system.
- Insufficient control of access to other GUARDIAN 90 objects. There is no protection of terminals, tapes, lines, and processes.
- No authorization auditing. There is no capability to record the granting or denying of access to an object.
- No auditing of changes to the security data base. Changes of object ownership and security are not recorded.

- No authentication auditing. Tandem does not support any means of auditing authentication attempts.
- Limited password management.
- No file creation control.

SAFEGUARD Protection

SAFEGUARD system protection software coexists with and extends GUARDIAN 90 protection by providing more extensive and general authentication, authorization, and auditing services.

Authentication

The first important service provided by SAFEGUARD is authentication, i.e., proving that users are who they say they are. While there are many new exotic technologies, such as retinal eye scanners and hand geometry and fingerprint readers, knowledge of a password, the method of authentication used by the GUARDIAN 90 operating system, remains the least expensive and most widely accepted means of identification.

Controlling the authentication mechanism is important. SAFEGUARD's capabilities include:

- Forcing a password to change periodically.
- Requiring a minimal password length.
- Requiring one-way encryption of passwords.
- Allowing a system manager to grant a user temporary access to a system by defining a user expiration date.
- Temporarily suspending a user's ability to access the system.

Perhaps most importantly, the SAFEGUARD system protection extends the concept of object ownership to also apply to users. With this mechanism, a site can define all users as belonging to a local user (e.g., the local security administrator) or as belonging to a network user (e.g., a single security administrator for the entire distributed system). It might also make sense to have some important users under the control of a local owner, while for ease of use the rest could belong to a network owner. SAFEGUARD provides a mechanism that allows a site to implement a variety of security policies.

Authorization

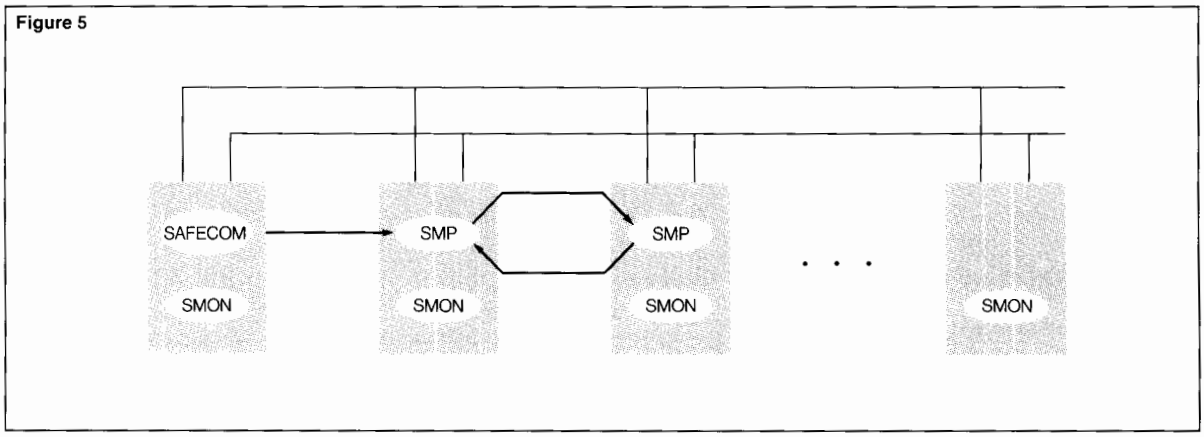
Once the user is authenticated, SAFEGUARD checks all requests for system objects based on a list of authorized users. The access list may contain local users, network users, local groups, or network groups. In addition SAFEGUARD can explicitly deny a user, or group of users, access to a particular object.

Objects that can be protected include disk files; terminals; encryption devices; SNA ports; X.25 communication lines; named processes; printers; and 6100 subsystem, TIL™, THL™, and INFOSAT™ devices. In addition, the SAFEGUARD system provides control over disk file creation. This is particularly useful in determining which users can place files on the $SYSTEM.SYSTEM and $SYSTEM.SYSnn subvolumes. Without this control, any user could introduce a program file that could masquerade as a system utility. Creation control is also extended to named processes. This can be very useful in preventing the creation of a masquerading process.

The main focus of the GUARDIAN 90 operating system is the protection of disk files.

The SAFEGUARD security system provides a smooth migration path from GUARDIAN 90 protection to SAFEGUARD protection by allowing the two forms of protection to coexist. Therefore, a user may choose to protect some critical disk files with SAFEGUARD and allow the remaining disk files to be protected by current GUARDIAN 90 security. In addition, SAFEGUARD can exist in a mixed EXPAND™ network with some nodes protected by SAFEGUARD and some not.

Figure 5.
SAFEGUARD processes.



As with users, SAFEGUARD objects are also owned. An object may either be owned by a local user or a network user. When an object is owned by a network user, all aspects of the object's security can be controlled from a remote node. Object owners can also temporarily suspend a user's ability to access their object. When the access authorities granted to users with the object's access list are suspended, only the object's owners, the owner's group manager, and the local SUPER ID are allowed access to the object.

Auditing

While authentication and authorization services are sufficient to provide on-node protection, auditing of these two activities is necessary to aid in detecting and tracking any attempts at breaking the protection mechanism. The SAFEGUARD protection system provides the capability to audit authentication requests, authorization requests, requests to modify the authentication data base (e.g., to change the user expiration date), and requests to modify the authorization data base (e.g., to add an entry to the access list). Since auditing is a time-consuming operation, SAFEGUARD software allows the user to choose whether only successful requests, only failed requests, or both are audited. This same degree of control is also applied to whether the request is local or remote.

Both of the authentication and authorization audit data bases are entry-sequenced files formatted so that ENFORM™ reports can be written. The audit records contain such information as the time of the attempt and who made it, as well as the node from which the attempt was made.

Implementation

The SAFEGUARD system is composed of a family of cooperating processes and follows the process model established for the Transaction Monitoring Facility (TMF™) and INSPECT™. It has three types of processes: SAFECOM, the SAFEGUARD Management Process (SMP), and the SAFEGUARD Monitor (SMON). Their relationship is shown in Figure 5. SAFECOM provides the interactive interface for SAFEGUARD. The command language is similar to the language used in other Tandem products such as TMFCOM and PATHCOM. SAFECOM is used to control both the authentication and authorization data bases throughout the network.

The SMP is a NonStop system process-pair responsible for maintaining the availability of the SMON in each CPU of a system protected by SAFEGUARD. The SMP also modifies the authentication and authorization data base and enforces the ownership rules. SAFECOM, for instance, talks to the SMP in order to provide the interactive interface for modifying security information. Finally, the SMP provides the authentication services.

The server processes (SMONs) are responsible for enforcing the authorization service for GUARDIAN 90 objects. One SMON runs in each CPU of a system protected by SAFEGUARD. Implementing one process per CPU provides both multi-CPU failure tolerance and improved performance over the use of NonStop process-pairs.

The SAFEGUARD authorization and authorization auditing service is supported locally with interfaces in the procedures OPEN, CREATE, RENAME, PURGE, NEWPROCESS, and STOP, and remotely with an interface in the EXPAND line handler. Authorization requests for a given object are routed to the SMON in the primary CPU for that device. For example, if the primary disk process for $DATA is located in CPU 4 then all requests to open a disk file on $DATA are handled by the SMON in CPU 4. This is true both remotely and locally. Furthermore, the authorization data base for $DATA is maintained on $DATA. For all system objects that do not have long-term media storage associated with them (e.g., processes, tape drives, communication devices, and terminals), the authorization data base is maintained on $SYSTEM.

Conclusion

Providing distributed system protection so that users can implement a variety of security policies requires both on-system and off-system mechanisms. Off-system protection is provided by encryption, while on-system protection is made possible by a combination of the hardware and the operating system.

The NonStop II, TXP, and EXT hardware and the GUARDIAN 90 operating system and its extension, SAFEGUARD, provide that protection in Tandem NonStop systems.

References

Data Encryption Standard. 1977. FIPS Publication 46. Department of Commerce.

Ehrsam, W.F., Matyas, S.M, Meyer, C.H., and Tuchman, W.L. 1978. A Cryptographic Key Management Scheme for Implementing the Data Encryption Standard. *IBM Systems Journal*, Vol. 17, No. 2, pp. 106-124.

Feistel, H. 1973. Cryptography and Computer Privacy. *Scientific American* 228, No. 5, pp. 15-23.

Lampson, B.W. 1971. Protection. In *Proceedings* of the Fifth Princeton Symposium on Information Sciences and Systems, Princeton University. Reprinted in *Operating System Review*, Vol. 8, No. 1.

Saltzer, J.H., and Shroeder, M.D. 1975. The Protection of Information in Computer Systems. In *Proceedings* of IEEE, Vol. 63, No. 9.

Popek, G.J. 1974. Protection Structures. *Computer*. Vol. 7, No. 6.

Wilkes, M. 1972. *Time Sharing Computer Systems*. American-Elsevier, Second Edition.

Acknowledgments

The writer acknowledges Wendy Bartlett, Leland Fong, Dave Lilja, Kevin Weigler, Roy Capaldo, Matt Matthews, Garry Easop, and Kevin Coughlin for their many contributions to this article.

Timothy Chou joined Tandem Software Development in May 1981. Prior to this, he completed a Ph.D. in Electrical Engineering at the University of Illinois, Urbana-Champaign.

PATHWAY IDS: A Message-level Interface to Devices and Processes

Intelligent Device Support (IDS) for the PATHWAY transaction processing system was first made available with the B10 release of the GUARDIAN™ operating system. IDS allows applications using PATHWAY to communicate with systems, devices, and processes directly, without the need for special front-end processes. IDS permits programs written in SCREEN COBOL to communicate with so-called intelligent devices (such as personal computers, automated tellers or ATMs, and point-of-sale machines) and GUARDIAN processes in which screen presentation will be, or already has been, performed (e.g., IBM 3270 passthrough).[1]

[1]This article was written before the B30 software release. Consult the most current manuals for full, current information about PATHWAY IDS, its scope and functionality.

This article begins with a brief history of the PATHWAY transaction processing system and then presents the following about IDS:

- Functional overview.
- New data definitions.
- New SEND MESSAGE statement.
- Functional overview of message processing.
- Terminal Control Process (TCP) resource consumption and configuration.
- Conversion procedures for users.
- Sample user-conversion program for users.
- Sample IDS program.

History of the PATHWAY System

To understand the role of IDS in the PATHWAY transaction processing environment, it is useful to retrace the evolution of the PATHWAY system and compare the objectives of the original product with its present direction. Until the introduction of its conversational features, the PATHWAY system was designed to serve conventional on-line applications and to provide communication and presentation services for a limited number of block-mode terminal types (6510, 6520, 6530, 6540, and IBM 3270). The data exchange that takes place between the PATHWAY system and these terminals is terminal-dependent and contains screen-specific information. The conversational version of PATHWAY was added to support TTY devices that do not operate in block mode.

To aid the PATHWAY system in supporting a wide variety of terminal types, Tandem programmers and Tandem users have written multithreaded, fault-tolerant front-end processes (FEPs). Their purpose is to translate the message traffic between the PATHWAY TCP and an unsupported device into a format acceptable to either. One popular example of this is the Terminal Dependency Eliminator (TDE), written by a Tandem analyst and made generally available to Tandem users.

Because of the Tandem NonStop system's excellent networking capabilities, an increasing number of computer users have chosen it for connectivity, device management, and device switching between application systems. Two examples of this are the controlling of large networks of ATMs and the providing of passthrough capabilities to applications running on hosts other than the Tandem NonStop system. The requirements of such communications-oriented and "hybrid" environments are reflected in the direction the development of the PATHWAY system has taken.

IDS carries PATHWAY's communications capabilities a step further by providing features that eliminate or reduce the need for the FEPs and provide an integrated interface to previously unsupported devices.

Functional Overview of IDS

IDS is best described as a message equivalent to the familiar block-mode ACCEPT and DISPLAY statements. In conventional block-mode SCREEN COBOL, working-storage data is mapped through the Screen Section with the ACCEPT and DISPLAY statements. In IDS, working-storage data is optionally mapped through a Message Section with a SEND MESSAGE statement.

As with Screen Section data items, user-conversion procedures can be invoked for elementary field-level items in the Message Section. In addition, message-level conversion is provided through the USER CONVERSION clause of the SEND MESSAGE statement.

New Data Definitions

For sending information to processes or devices that do not require screen formats, new IDS data-definition constructs in the Data Division are provided to distinguish message structures from screen structures. The section is identified by a Message Section statement. This section does not consume terminal context area; it provides a mapping function for data from a terminal to working-storage data elements (and vice versa).

Message Section Message-description Entry
A number of messages can be defined in the Message Section with the conventional SCREEN COBOL level numbers. Level 01 identifies the beginning of a message, and subordinate group and elementary data items can be defined. A message can be as large as 12 Kbytes (12,288 bytes). (Software releases before the B30 release support only an 01-level alphanumeric data item.)

Field-characteristic Clauses
Four field-characteristic clauses are possible in the Message Section:

- MESSAGE FORMAT.
- PICTURE.
- TO, FROM, and USING.
- USER CONVERSION.

The PICTURE; TO, FROM, and USING; and USER CONVERSION clauses are the same as defined in the Screen Section. The MESSAGE FORMAT clause provides for three types of message format: FIXED, VARYING1, and VARYING2. The MESSAGE FORMAT clause can only be used on the 01-level definition.

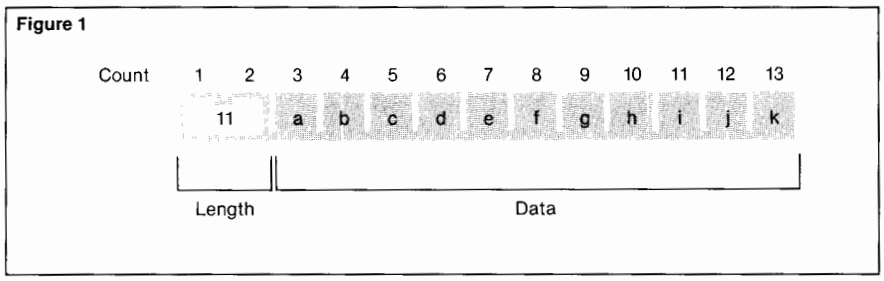


Figure 1.

The format of a variable-length message. The format consists of a 1- or 2-byte length field and the data portion. The length field indicates the count of bytes in the data portion of the message, not including the length field itself. In this VARYING2 message, the length is 11 as there are 11 data characters following the length descriptor.

Data Editing and Transformation

The data is transformed between the paired Working-storage and Message Section PICTURE clauses by normal SCREEN COBOL editing and move rules. The PICTURE clauses must match in type and may differ in length (and, for numeric items, in scale). Via the Message Section, data can be gathered from and scattered to discontiguous data items in working storage.

Variable-length Messages

The format of a variable-length message consists of a 1- or 2-byte *length field* and the data portion. The length field provides a binary count of the bytes in the data portion of the message, not including the length field itself.

Figure 2

```
SEND MESSAGE    { send-message                }
                { [send-message] reply-spec}

                [ USER [CONVERSION] numeric-literal ]
                [ TIMEOUT                timeout-value ]
                [ ON ERROR      imperative-statement ]
  reply-spec
      REPLY [ CODE FIELD [IS] code-field]
      {                                  YIELDS reply-message    }
      {{CODE reply-code [,reply-code]..YIELDS reply-message}...}
```

Figure 2

An abbreviated syntax description of the SEND MESSAGE statement. With this statement, a program can send a message, receive a reply, or send a message and *receive a reply.*

Figure 1 is an example of a VARYING2 message. For this message, the length is 11 as there are 11 data characters following the length descriptor. If the length field were to contain 13, the TCP would report an error, saying the message was too short. The VARYING1 length can describe a message up to 255 bytes long. The VARYING2 format can hold a maximum length value of 64 Kbytes, but the maximum message length is 12 Kbytes.

New SEND MESSAGE Statement

The SEND MESSAGE statement is syntactically similar to the SEND statement that is used to communicate with servers. This means that an intelligent device can reply to a message with one of a set of replies and have that reply identified and processed according to its format. This is in contrast to the ACCEPT statement, in which a single reply is always paired to a screen image. The abbreviated syntax of the SEND MESSAGE statement is described in Figure 2. With this statement, a program can send a message, receive a reply, or send a message *and* receive a reply.

The simplest type of reply is expressed as

REPLY YIELDS reply-message

where only one reply format is expected. This is functionally equivalent to the ACCEPT statement. If multiple reply formats are to be received, the CODE reply-code clause must be used to identify a specific reply-message. One or more reply-codes may identify the same reply-message. If the reply-code is not in the first 2 bytes of the reply-message, the CODE FIELD clause must be used. The code-field is expected to be at the same location and of the same size for all reply-message formats. Like the server SEND statement, the TERMINATION-STATUS special register is set as an index to the reply-message received upon normal termination or as an error number if the ON ERROR clause is executed. The TERMINATION-SUBSTATUS may also be set, depending on the error condition.

Functional Overview of Message Processing

Two methods of sending and receiving messages exist, and user-conversion procedures can be invoked (optionally) on either method to examine or alter the message contents. A message can be:

- Stored into or sent from working storage directly, with an optional USER CONVERSION clause on the SEND MESSAGE statement.
- Stored into or sent from working storage via the Message Section, with an optional USER CONVERSION clause on the SEND MESSAGE statement and/or on the data item in the Message Section.

User-conversion procedures are discussed in a later section. The following sections assume that a USER CONVERSION clause is not specified.

Messages Directly into/from Working Storage

These messages are defined by the PICTURE clause or group definition and are essentially fixed-length messages. On input, the message must be equal in length to the working-storage item. On input or output, the message can be "variable" when the data item is described with an OCCURS DEPENDING ON clause. Data is moved directly from the terminal I/O buffer into working storage or vice versa. (See Figure 3.)

Messages into/from Working Storage via the Message Section

When messages are stored into or sent from working storage using the Message Section and an intermediate-field work area, the PICTURE clause(s) on the Message Section elementary items define the send-message and reply-message structures. (See Figure 4.) The OCCURS DEPENDING ON clause for a working-storage data item cannot be used because there is no equivalent clause in the Message Section.

The MESSAGE FORMAT clause applies to the entire message and is currently only specifiable at the 01 level. Messages can be fixed or variable in length. Variable-length messages are defined with the MESSAGE FORMAT VARYING1 or MESSAGE FORMAT VARYING2 clauses. The VARYING1 or VARYING2 length field is inserted and removed by the TCP during message processing and is not available to the program in working storage.

Figure 3

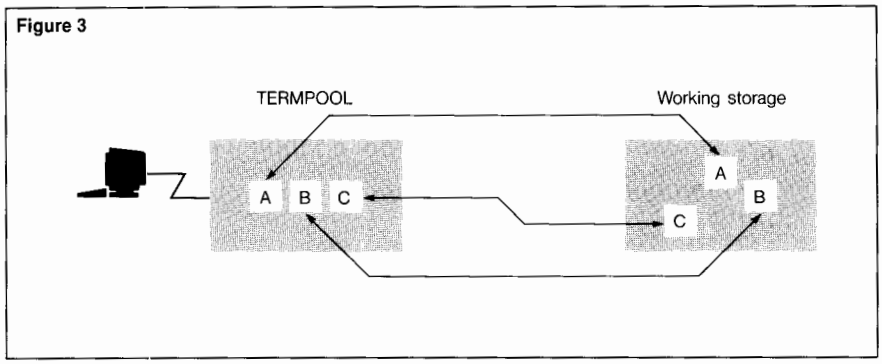


Figure 4

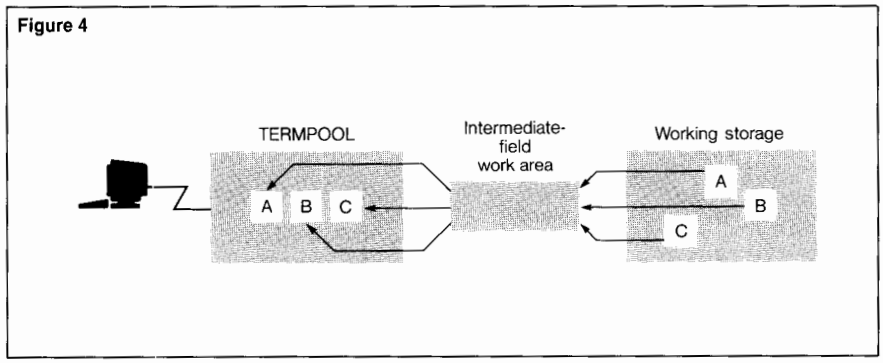


Figure 3.
Field transfer directly from TERMPOOL into working storage and vice versa with no Message Section, or from TERMPOOL into working storage with a Message Section, with or without a USER CONVERSION clause specified on the elementary data items.

Figure 4.
Field transfer via an intermediate-field work area to TERMPOOL from working storage, using a Message Section with or without a USER CONVERSION clause specified on the elementary data items.

Figure 5

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. IDS-EXAMPLE.
ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
    SOURCE-COMPUTER. T16.
    OBJECT-COMPUTER. T16, TERMINAL IS INTELLIGENT.
DATA DIVISION.
  WORKING-STORAGE SECTION.
* The data item WS-MESSAGE is referenced in the Message Section.
* Note that the reply-code is not required to be at offset 0 of the
* reply-message, nor is it required to be numeric.
  01  WS-MESSAGE                  PIC X(20) VALUE "Send back a message ".
  01  WS-REPLY.
      05  WS-REPLY-LENGTH         PIC 9(4) COMP.
      05  WS-REPLY-CODE           PIC X(02).
      05  WS-REPLY-BODY           PIC X(1920).
  01  WS-SPACE-ALLOC              PIC X VALUE " ".
  01  WS-TERMINATION-STATUS       PIC 9(04) COMP.
  01  WS-TERMINATION-SUBSTATUS    PIC 9(04) COMP.
MESSAGE SECTION.
  01  MS-MESSAGE                  MESSAGE FORMAT IS FIXED.
      05  MS-MESSAGE-HEADER       PIC X(25)
                                  FROM WS-SPACE-ALLOC.
      05  MS-MESSAGE-BODY         PIC X(20)
                                  FROM WS-MESSAGE.
  01  MS-REPLY                    MESSAGE FORMAT IS FIXED.
      05  MS-REPLY-LENGTH         PIC 9(4) COMP
                                  TO WS-REPLY-LENGTH.
      05  MS-REPLY-CODE           PIC X(02)
                                  TO WS-REPLY-CODE.
      05  MS-REPLY-BODY           PIC X(960)
                                  TO WS-REPLY-BODY.
  01  MS-DUMMY-REPLY              MESSAGE FORMAT IS FIXED.
                                  PIC X(2048)
                                  TO WS-SPACE-ALLOC.
PROCEDURE DIVISION.
MAIN SECTION.
    PERFORM 100-SEND-MESSAGE.
010-EXIT.
    EXIT PROGRAM.
* The outbound message is sent via a Message Section structure.
* This structure inserts a place holder field for a header data item
* to be provided by the user-conversion procedure.
* The inbound message "AA" is placed directly into working storage.
* The "BB" is transformed via the user-conversion routine that
* modifies the message by storing the message length in the message.
* The dummy reply "XX" is provided to lengthen the intermediate work
* area for the user-conversion procedure in processing the reply.
100-SEND-MESSAGE.
  SEND MESSAGE MS-MESSAGE
    REPLY CODE FIELD IS WS-REPLY-CODE
      CODE "AA" YIELDS WS-REPLY
      CODE "BB" YIELDS MS-REPLY
      CODE "XX" YIELDS MS-DUMMY-REPLY
    USER CONVERSION 1
    ON ERROR
      PERFORM 900-SENDMSG-ERROR.
* A terminal defined as intelligent is likely to be dependent on a
* PATHWAY server class to log its errors.
900-SENDMSG-ERROR.
  MOVE TERMINATION-STATUS
    TO WS-TERMINATION-STATUS.
  MOVE TERMINATION-SUBSTATUS
    TO WS-TERMINATION-SUBSTATUS.
  SEND WS-TERMINATION-STATUS
       WS-TERMINATION-SUBSTATUS
    TO "IDS-ERROR-SVR" .
```

To pass the length field between working storage and a user-conversion procedure, include an elementary data item in the Message Section data definitions. See the data item

```
05 MS-REPLY-LENGTH   PIC 9(4) COMP
   TO WS-REPLY-LENGTH.
```

in Figure 5. The annotated SCREEN COBOL code in this figure represents a typical implementation of a Procedure Division for IDS. The objective of the program is to request raw data from a program running in an intelligent device and store it in working storage. In this example, protocol-related overhead must be added to the send-message and stripped from the reply-message. In order to do this, the SEND MESSAGE statement invokes a user-conversion procedure.

On input, the user-conversion procedure must shift the data portion of the message 2 bytes to duplicate length value (VARYING1 must be converted to 2 bytes for the COMP usage). Although the example doesn't do this, on output, the length may be passed to the user-conversion procedure in the message, used, and then shifted off. For variable-length messages, the TCP automatically truncates any trailing blanks and inserts the varying field length in front of the message.

Figure 5

A typical Procedure Division for IDS. The program requests raw data from a program running in an intelligent device and stores it in working storage. A user-conversion procedure is invoked from the SEND MESSAGE statement to add protocol-related overhead to the send-message and to modify one of the reply-messages.

TCP Resource Consumption and Configuration

To perform the data transfer and formatting operations required for IDS, two or three data areas are required. The basic data areas are working storage and the terminal buffer area. Working storage is defined by the PICTURE clause for an elementary data item, and for a group item, it is the sum of these elementary items. The terminal buffer area in TERMPOOL is allocated according to the larger size of the send-message or the largest of the reply-messages plus one.

An additional intermediate work area is required when data is to be formatted between the Message Section and working storage (Figure 4) and/or when a USER CONVERSION clause is used on the SEND MESSAGE statement (Figure 6). This area is allocated at the end of the current terminal context area. The size of the total area is specified with the SET TCP MAXTERMDATA parameter in PATHCOM. (PATHCOM is the command interface to PATHMON, the central control process in the PATHWAY system.) The value can be empirically derived during unit testing via the PATHWAY STATS command to determine the largest area ever allocated during the execution of the application. (See Wong, 1984, for more information on the PATHWAY TCP and PATHWAY statistics.)

User-conversion Procedures

When there is a requirement for bit-manipulation functions, communication or device control characters, or variable-length messages that do not conform to the VARYING1 or VARYING2 formats, user-conversion procedures are necessary. Four user-conversion procedures provide for the reformatting of messages or individual numeric and alphanumeric fields on both input and output operations. (See Appendix D of the *PATHWAY SCREEN COBOL Reference Manual* for details.) User-conversion procedures can be called at two levels during message processing.

Send-message Processing

When a send-message is constructed, one or more working-storage fields are moved to an area allocated in the TCP's TERMPOOL (Figure 3 or 4). Each field can be reformatted at this time by a user-conversion procedure (specified on a data item in the Message Section). The procedure is given an intermediate work area in which to store the results of the conversion. The TCP moves the resultant field to the buffer in TERMPOOL (Figure 4).

After the send-message is built, the message can be reformatted by a user-conversion procedure (e.g., adding control characters or a routing header) specified on the SEND MESSAGE statement. Again, an intermediate work area is provided that is large enough to hold the entire message. The TCP moves the message back to TERMPOOL for the length specified (Figure 6). To lengthen a send-message (i.e., to add header and/or trailer fields), the Message Section message-description entry must contain additional "place holder" fields. For example, in Figure 5, the statement

```
05 MS-MESSAGE-HEADER  PIC X(25)
    FROM WS-SPACE-ALLOC.
```

would cause 25 bytes to be reserved in the send-message for a header, while only consuming one character in working storage.

Figure 6

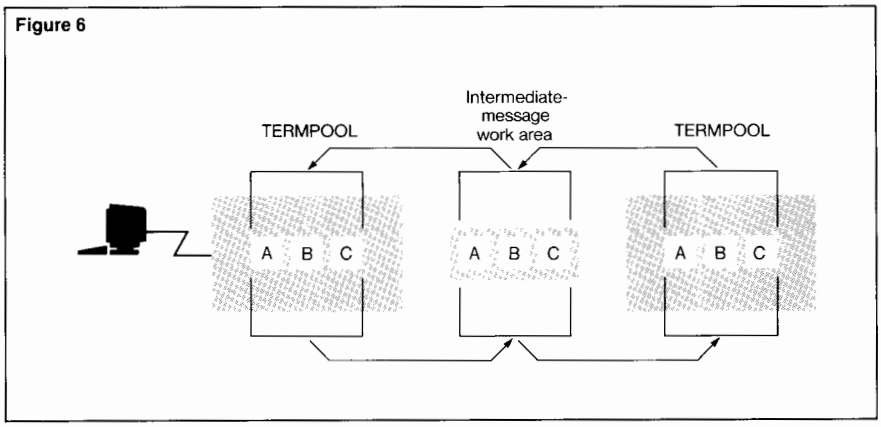


Figure 6

Message transfer via an intermediate-message work area when a USER CONVERSION clause is specified on a SEND MESSAGE statement.

Figure 7

```
PROC USER^ALPHA^INPUT^MSG^CONV ( USERCODE, ERROR, INPUT,
                                 INPUT^LEN, INTERNAL, INTERNAL^LEN,
                                 FILL^CHAR, FILL^OFF, RIGHT^JUSTIFIED,
                                 FIELD^RETURNED, FIELD^PRESENT );
    INT      USERCODE;              ! Set by TCP
    INT      .ERROR;                ! Set by user procedure
    STRING   .EXT INPUT;            ! Set by TCP
    INT      .INPUT^LEN;            ! Set by TCP; modified by user
    STRING   .EXT INTERNAL;         ! Set by user procedure
    INT      INTERNAL^LEN;          ! Set by TCP
    STRING   FILL^CHAR;             ! Set by TCP
    INT      FILL^OFF;              ! Set by TCP
    INT      RIGHT^JUSTIFIED;       ! Set by TCP
    INT      .FIELD^RETURNED;       ! Set by user procedure
    INT      .FIELD^PRESENT;        ! Set by user procedure
    Begin
      String    .ext p;             ! Temporary pointer
      Int       Length := Internal^len - 2d;  ! Stopper for internal^len
      Case usercode of
        Begin
    !0!                             ! Routine 0 - Change numbers to "*"
          Begin                     ! Perform conversion
            @p := @internal; ! reply-code.
            @internal := @internal + 2d;
            While length do
              Begin
                If $numeric(internal) then
                  Internal ':=' "*";
                @internal := @internal + 1d;
                Length := length - 1;
              End;
            @internal := @p;
            Error := 0;
          End;
    !1!                             ! Routine 1 - Change letters to "#"
          Begin                     ! Perform conversion
            @p := @internal;
            @internal := @internal + 2d;
            While length do
              Begin
                If $alpha(internal) then
                  Internal ':=' "#";
                @internal := @internal + 1d;
                Length := length - 1;
              End;
            @internal := @p;
            Error := 0;
          End;
    !2!                             ! Routine 2 -
          Begin                     ! No conversion - NOP
          End;
    !>2!                            ! Any other routine is an error
          Begin                     !
          Error := 105;             ! Return error code
          End;
        End;                        ! End Case
    End;                            ! End Proc
```

Figure 7

An example of a case structure that best implements a user-conversion procedure.

Reply-message Processing

When a reply-message is received, the message can be reformatted by a user-conversion procedure (e.g., removing control characters or a routing header) specified on the SEND MESSAGE statement. The procedure is given an intermediate work area large enough to store the largest reply (Figure 6). The TCP moves the message back to TERMPOOL for the length specified. To cause a larger allocation, the SEND MESSAGE statement would have to include a "dummy" message-description entry that described the desired length. See Figure 5 for the Message Section statement

```
01 MS-DUMMY-REPLY  PIC X(2048)
    TO WS-SPACE-ALLOC.
```

and its use in the SEND MESSAGE statement.

Next, the reply-code tests are made to select a reply-message format that is then used to move the field(s) to working storage. Each field can be reformatted at this time by a user-conversion procedure (specified on a data item in the Message Section). The procedure is given the address and length of the working-storage data item (Figure 3).

User-conversion Procedure Names

During reply-message or send-message operations, the respective user-conversion procedures USER^ALPHA^INPUT^MSG^CONV or USER^ALPHA^OUTPUT^MSG^CONV may be called for an alphanumeric field in the Message Section or, on a SEND MESSAGE statement, for the entire message, which is treated as an alphanumeric group item even if it contains a single numeric field. The procedures USER^NUMERIC^INPUT^MSG^CONV or USER^NUMERIC^OUTPUT^MSG^CONV may be called for only a numeric field in the Message Section.

Note: The user-conversion procedure must be *fully* tested to ensure that no data-formatting errors would occur to cause the terminal context area to be exceeded. This would affect another terminal's data space and might make debugging very difficult.

Numbering User-conversion Procedures

When numbering user-conversion routines, programmers should follow certain conventions. If a USER CONVERSION clause is specified on the USING clause or on the SEND MESSAGE statement, there must be a pair of routines in the output *and* input procedures that handle the same number. If processing is to be done on only one direction (e.g., the message is reformatted on output), the other procedure must contain an equivalent routine, which can be nothing more than a BEGIN-END statement pair (effectively a no-operation routine).

Note: The TCP performs the data transfer based on the PICTURE clauses before the user-conversion procedure is called, in case the procedure is a no-operation routine.

Sample User-conversion Procedure

Writing an IDS user-conversion procedure is no different from writing a procedure to convert terminal-screen data or support 3270 attention keys. The *PATHWAY SCREEN COBOL Reference Manual* provides a detailed catalog of input and output parameter requirements for the routines. The example in Figure 7 contains the case structure that implements a user-conversion procedure.

Conclusion

Intelligent Device Support, or IDS, enhances the PATHWAY transaction processing system by providing language elements in SCREEN COBOL that process messages to intelligent devices, in addition to formatted screen terminals. The new SEND MESSAGE statement adds power to device handling such that a single message can be sent and any one of a number of replies can be processed in a single statement. These messages can now be fixed or variable in length.

In addition, user-conversion routines can be written at both the field and message levels to format data at a lower level than that provided in SCREEN COBOL. As PATHWAY now provides a fully integrated solution to the support of intelligent devices within a single TCP process, the need for user-written front-end processes should be greatly reduced.

References

PATHWAY SCREEN COBOL Reference Manual. Part no. 82424 A00. Tandem Computers Incorporated.

Wong, R. 1984. A New Design for the PATHWAY TCP. *Tandem Journal*. Vol. 2, No. 2. Tandem Computers Incorporated.

_____. 1984. Understanding PATHWAY Statistics. *Tandem Journal*. Vol. 2, No. 2. Tandem Computers Incorporated.

Acknowledgments

The authors would like to thank Bill Firestone, Jim Gateley, and Bob Vannucci of Software Development for their technical review.

Mark Anderton is a staff analyst in the Application Design Support Group of Large Systems Support. His area of expertise is the ENCOMPASS distributed data management system, including the PATHWAY transaction processing system, TMF, and other data-base products. Mark joined Tandem Manufacturing MIS in 1980 where he worked as a programmer and, later, as a programming manager.

Mike Noonan is an advisory analyst in the Application Design Support Group of Large Systems Support. He supports ENCOMPASS products. Mike joined Tandem in 1981, following 13 years as a systems and applications programmer of on-line transaction processing and batch systems. Mike has a business degree from the University of Minnesota.

New TAL Features

A new, enhanced version of TAL™, the Tandem Application Language, is available with the B20 software release. This article discusses the following new features:

- Labeled CASE statement.
- Structure improvements (including pointers in structures and substructure declarations).
- Automatic allocation of extended memory.
- Unsigned data types.
- Literal declarations.
- MOVE statement improvements.
- Errorfile directive.

Labeled CASE Statement

The CASE statement provides a multiway control structure in which the value of an index expression is used to transfer control to one of several case branches.

Before the B20 release, the TAL CASE statement required users to express the correspondence between the value of an index expression and case branch implicitly, according to the sequential ordering of each of the case branches. For example:

```
PROC calc(op, a, b, c);
INT op, a, b, .c;
BEGIN
CASE op OF
      BEGIN
      GOTO no^op;     ! selected if  op = 0 !
      c := a + b;     !    "    "    op = 1 !
      c := a - b;     !    "    "    op = 2 !
      GOTO no^op;     !    "    "    op = 3 !
      c := a * b;     !    "    "    op = 4 !
      c := a / b;     !    "    "    op = 5 !
      ;               !    "    "    op = 6 !
      c := a + b;     !    "    "    op = 7 !
OTHERWISE
      no^op:
            IF op <>0 THEN
                CALL error(op);
      END;
END; ! calc !
```

Note the implicit, positional correspondence between index expression, the value of op, and case branch. Besides being error-prone, especially for large numbers of case branches, this positional scheme imposes some awkward

coding when the same case branch is to be selected by more than one value of the index expression. Either a GOTO must be coded, as shown by index values 0 and 3, or case branches must be duplicated, as shown by index values 1 and 7.

In the B20 release, CASE statement branches may be labeled. The principal improvement is that the correspondence between the index value and case branch is expressed with explicit, nonpositional syntax. In this new form, the example becomes:

```
PROC calc(op, a, b, c);
INT op, a, b, .c;
BEGIN
CASE op OF
      BEGIN
6 ->
      ;                    ! selected if  op  =  6 !
4 ->
      c := a  *  b; !   "    "   op  =  4 !
1, 7 ->
      c := a  +  b; !   "    "   op  =  1 !
2 ->
      c := a  -  b; !   "    "   op  =  2 !
5 ->
      c := a  /  b; !   "    "   op  =  5 !
OTHERWISE ->
      IF op <>0 THEN
           CALL error(op);
      END;
END; ! calc !
```

As this example shows, case branches can be coded in any order since the index/branch correspondence is given by the "label." Note also that "index gaps," such as the values 0 or 3, do not have to be coded as GOTOs; because they are unspecified, they both automatically branch to the OTHERWISE clause. Finally, note how simply a single case branch can be made to correspond to more than one index value, as shown by values 1 and 7.

The syntax[1] for the labeled CASE statement is given in Figure 1. The following examples highlight some additional features.

Figure 1

```
CASE <selector> OF
     BEGIN
     <alternative>;
     [ <alternative>; ]
          .
          .
          .
     [ <alternative>; ]
     [ OTHERWISE -> <statement-s>; ]
     END
where
     <selector>
     is an INT arithmetic expression that uniquely selects the case
     <alternative> to be executed.

     <alternative> is
          <case label> [, <case label>] ... -> <statement-s>
     An <alternative> is a sequence of <statement>s, <statement-s>, and
     an associated set of constant INT values, as specified by <case label>s.
     The <statement>s of <alternative> are executed if <selector> equals
     one of its associated values.
     No two <alternative>s can have an associated value in common.

     <case label> is
          <INT constant> or <INT constant-a>.. <INT constant-b>
     <case label>s are used to specify constant INT values. The first form
     specifies a single value and the second specifies all INT values i such that
     <INT constant-a> <= i <= <INT constant-b>. When used, the second
     form must specify at least one value.

     <statement-s> is
          <statement> [; <statement>] ...
     <statement> can be any TAL statement, labeled or unlabeled.

     OTHERWISE ->
     specifies an optional sequence of statements to be executed if no
     <alternative> is selected by <selector>.
     If this clause is not used and no <alternative> is selected, an instruction
     fault will occur.
```

Figure 1.
CASE syntax. TAL now includes a labeled CASE statement.

In a labeled CASE statement, consecutive selector (index) values that branch to the same alternative (case branch) need not be individually listed. Instead, they can be abbreviated as a range of values, from low value to high value:

```
CASE i OF
      BEGIN
0 -> ! alternative 0 ! ;
-10 .. -1 ->
      ! executed when -10 <= i <= -1 ! ;
1 .. 9 ->
      ! executed when 1 <= i <= 9 ! ;
      END;
```

[1]Syntax conventions are as follows: lowercase letters within angle brackets (< >) represent all user-supplied variable entries; braces ({ }) indicate that exactly one of the options listed must be selected; brackets ([ ]) indicate that the field is optional and any number, including 0, of the enclosed options may be chosen; an ellipsis (...) immediately following a pair of brackets or braces indicates that the enclosed syntax can be repeated any number of times.

Case label ranges and single case labels can be used in combination to provide a very flexible means of selecting case alternatives:

```
CASE i OF
      BEGIN
1, 3, 11 ->
            ! Executed when i = 1 OR !
            !       i = 3 ORi = 11   !
      i := i * 3;
      i := i * 11;
            ! Note that a labeled case
            ! alternative can consist of
            ! a sequence of statements,
            ! unlike the unlabeled case
            ! branch.                    !
0, 2, 4 .. 10 ->
      ;     ! Executed when i = 0 OR !
            !       i = 2          OR !
            !       ( 4 < = i < = 10.)!
        END;
```

Within a labeled case statement, each case label can be specified (singly or in a range) only once:

```
CASE i OF
      BEGIN
0 -> ;
-1 .. 2 ->;     ! ILLEGAL, 0 has already     !
                ! been specified.            !
3, 3 ->;        ! ILLEGAL, even in the !
                ! same case label list.  !
4 ->;
5, 4 ->;        ! ILLEGAL !
   END;
```

Finally, note that the labeled CASE statement provides a predictable, more graceful handling of an out-of-bounds selector expression when no OTHERWISE clause is coded. If arithmetic traps are enabled, the CASE statement generates an overflow trap; if traps are disabled, control passes to the end of the statement. While unpredictable branches were possible with the unlabeled CASE statement, they are no longer possible when the labeled CASE statement is used.

Structure Improvements

Pointers in Structures

Before the B20 release, dependencies between different data objects could not be easily expressed in TAL. This was primarily because pointer variables (variables containing the addresses of other variables) could not be declared as component fields of structures and substructures.

Users could work around this restriction by representing pointer fields within a structure as INT or INT(32) fields. Accessing the target of such a pointer was quite awkward because users had to copy the integer field into a suitably declared temporary pointer before they could access the target data.

In the B20 release, TAL solves this problem by an extension that allows the declaration of pointer fields within structures and substructures. In addition, pointer fields within a structure are "de-referenced" (indirectly accessed) in the same way as pointers outside of structures. As a result, the need for temporary pointers is greatly reduced.

Not surprisingly, the syntax for declaring pointers within a structure is very similar to the syntax for declaring pointers outside of structures. The only exception here is that pointer fields, like any other component field of a structure, cannot be initialized. See Figure 2 for the full syntax.

With this extension, defining linked data structures such as lists and binary trees is very straightforward, as shown in Figure 3.

BinaryNode is declared to be a structure template that can be used to declare structures with three component parts: a value field and two structure pointers that will contain the addresses of subordinate instances of the BinaryNode template.

As with pointers outside of structures, access to the value of a pointer is denoted with the "at" sign (@). The only difference is that a pointer variable must be fully qualified (as must any other reference to a structure field) with the name of its enclosing structure and all of its enclosing substructure.

With the exception of name qualification, a reference to the target of a pointer within a structure is similar to a reference to the target of a pointer outside a structure. The only difference is that a pointer within a structure cannot be subscripted.

While pointers in structures generally behave the way one might expect, there are some important limitations. First, the target of a structure pointer contained in a structure is not a substructure; it is itself a structure. This is reflected in the $OFFSET function: the offset of a field in the target excludes the offset of the structure pointer and the byte length of the pointer itself. For example:

$OFFSET(Parent.RightChild)

returns the byte offset of the RightChild of Parent; this value is 4.

$OFFSET(Parent.LeftChild.RightChild)

returns the byte offset of the RightChild of the target of the LeftChild of Parent; this value is also 4.

Second, pointers and structure pointers within a structure cannot be initialized at declaration time as is true of other structure items.

A third difference is that when a structure pointer within a structure is declared, the identifier used for the referral cannot be a forward reference. That identifier must have already been seen by the compiler, although its declaration may be incomplete at the point of reference. For example:

```
STRUCT me(*);
BEGIN
  INT .me^link(me);
    ! This is legal.
  INT .you^link(you);
    ! This would cause an undeclared
    ! identifier error.
  INT info [0:9];
END;
STRUCT you(*);
BEGIN
  INT .me^link(me);
    ! This is ok. TAL has seen "me."
  INT .you^link(you);
    ! This is also ok.
  INT different^info [0:19];
END;
```

Figure 2

```
<type> { .     } <identifier>
       { .SG   }
       { .EXT  }
                  [ = <previous item> ];
```

where <type> is STRING, INT, INT(32), FIXED, REAL, or REAL(64).

The syntax for structure pointer declarations within structures is

```
{ INT    } { .     } <identifier> ( <referral>)
{ STRING } { .SG   }
           { .EXT  }
                  [ = <previous item>];
```

The syntax for initializing a pointer within a structure is

```
@<struct-index> [[.<substruct-index>]...].<pointer-name>:=
        <arithmetic-expression>;
```

where <struct-index> [<substruct-index>] is the name of a structure [substructure] with or without an index expression.

Figure 3

```
STRUCT BinaryNode(*);
BEGIN
  INT NodeActive; ! < > 0 implies an active node.
  INT .LeftChild(BinaryNode);
  INT .RightChild(BinaryNode);
END;
STRUCT Parent(BinaryNode);
STRUCT NewChild(BinaryNode);
@Parent.LeftChild := @NewChild;
  ! NewChild is made the target of the
  ! LeftChild pointer in Parent.
@NewChild.LeftChild := Nil;   ! Initialize each of
@NewChild.RightChild := Nil; ! the pointers in NewChild
NewChild.NodeActive := -1;    ! to Nil (-1).
Parent.LeftChild.NodeActive := 0;
  ! Mark the target of LeftChild in
  ! Parent inactive.
@Parent.LeftChild := @Parent.LeftChild.LeftChild;
  ! Delete the target of LeftChild of Parent
  ! by making the LeftChild of target the
  ! new target of LeftChild of Parent. !
Parent.RightChild := Nil;
  ! Illegal because pointers in
  ! structures cannot be subscripted.!
```

Figure 2.

Syntax for pointer declarations within structures.

Figure 3.

In the B20 version of TAL, defining linked data structures such as lists and binary trees is straightforward. (Note how the "@" prefix in "@Parent.LeftChild. LeftChild" is used to access the address of the target of the LeftChild of the target of the LeftChild of Parent.)

Figure 4

```
STRUCT <identifier> (<referral>)
                [ "[" <lower-bound> : <upper-bound> "]" ]
                [ = <previous item> ];
```

<identifier>
is the name of the new substructure.

<referral>
is the name of a previously declared structure.

<lower-bound>
is a constant expression in the range -32768 through 32767 that specifies the first substructure occurrence for which to allocate storage. The default value is 0 (one occurrence). Each occurrence is one copy of the substructure.

<upper-bound>
is a constant expression in the range -32768 through 32767 that specifies the last substructure occurrence for which to allocate storage. The default value is 0 (one occurrence).

<previous item>
is a previously declared structure item or substructure on the same level as <identifier>.

Figure 4.
Syntax for a substructure declared via a referral.

Substructure Declarations

Before the B20 release, substructure declarations could only be shared through the use of a DEFINE.

In the B20 release, substructures can be declared through the use of referrals to previously declared structures or structure templates. This makes the treatment of substructures and structures more uniform and should improve the readability of code. For example:

```
STRUCT t(*);
BEGIN
  STRING a,b,c;
END;

STRUCT Sub^Dir;
  ! Direct declaration.
BEGIN
  STRING s;
  STRUCT ss1;
  BEGIN
    STRING a,b,c;
  END;
END;

STRUCT Sub^Ref;
  ! Referral declaration. !
BEGIN
  STRING s;
  STRUCT ss2(t);
END;
```

Code modularity is also improved if referrals are used whenever two or more structures have data in common or contain data that is organized in the same manner. This makes maintenance of large programs much easier for programmers. For example:

```
STRUCT customer^info(*);
BEGIN
  STRING name[0:39];
    ! Company name. !
  STRUCT address;
    ! Company address. !
  BEGIN
    STRING number^and^street[0:49];
    STRING city[0:19];
    STRING state[0:19];
    STRING zip[0:4];
  END;
END;

STRUCT .order^for;
  ! Order placed. !
BEGIN
  INT order^number;
  STRUCT customer(customer^info);
  INT part^number;
  INT quantity;
  STRING item^price [0:4];
  STRING total^price[0:6];
END;

STRUCT .customer^history;
  ! History file updated with each made. !
BEGIN
  STRUCT customer(customer^info);
  STRING last^year^sales[0:8];
    ! Last year's revenue.
  STRING year^to^date[0:8];
    ! This year's revenue to date.
END;
```

In this example, the template CUSTOMER^INFO is used in both ORDER^FORM and CUSTOMER^HISTORY. Changes made to any of CUSTOMER^INFO's fields would be propagated to those structures referring to it.

TAL allocates storage for a substructure declared via a referral in the same way that it allocates storage for a substructure declared via a body definition. (See Figure 4.) There are, however, the following exceptions.

First, a substructure declared via a referral always starts on an even-byte boundary. This rule is imposed so that substructures declared via a referral and all structures have the same alignment characteristics. In the first example,

$OFFSET(sub^dir.ss1) = 1,

but

$OFFSET(sub^ref.ss2) = 2.

A pad byte separates sub^ref.s and sub^ref.ss2.

A second exception is that $LEN for a substructure declared via a referral is always an even number. This rule is imposed so that two substructures declared with the same <referral> structure always have the same length. In the first example,

$LEN(sub^dir.ss1) = 3,

but

$LEN(sub^ref.ss2) = 4.

Substructure referrals may not be made to the name of the structure whose body is currently being declared. Such a recursive declaration would produce an infinite structure.

Automatic Allocation of Extended Memory

This enhancement provides easier access to extended memory in TAL by supporting automatic memory allocation for extended global arrays and structs.

Before the B20 release, users needed to manually allocate extended memory and initialize pointers to that space. (See Figure 5a.)

In the B20 release, for the new EXTENDED ARRAY and STRUCT declarations, the allocation and deallocation of an appropriately sized segment and the initialization of pointers to that space are made by the compiler, BINDER™, and the GUARDIAN 90 operating system, and are transparent to TAL users. (See Figure 5b.)

Figure 6 contains a formal syntax description of array and STRUCT declarations in extended memory.

Figure 5.
Allocation of extended memory. (a) Before the B20 release of TAL, users die this manuall. (b) In the B20 release, the TAL compiler, BIDER, and the GUARDIAN 90 operating system allocate it automatically.

Figure 6.
Syntax for extended arrays and STRUCTS.

Figure 5

(a)

```
?SOURCE $system.system.extdecs (ALLOCATESEGMENT, USESEGMENT,
?                               DEALLOCATESEGMENT)

STRING .EXT ba^ptr := 0D; ! Extended pointer to byte array.!
INT status := 1000;
LITERAL seg^id = 0;
LITERAL seg^len = 2048D; ! Size of segment allocated in bytes.!
LITERAL dealloc^flags = 1; ! For DEALLOCATESEGMENT later.!
INT old^seg^num := -1;

PROC ext^addr^example MAIN;
BEGIN
   status := ALLOCATESEGMENT (seg^id, seg^len);
   old^seg^num := USESEGMENT (seg^id);
   @ba^ptr := %2000000D; ! Initialize pointer to the start of the segment.
   ba^ptr ':=' "This is a sample string.";
   CALL DEALLOCATESEGMENT (seg^id, dealloc^flags);
END;
```

(b)

```
STRING .EXT ba^ptr [0:2047]; ! Extended byte array.!

PROC ext^addr^example MAIN;
BEGIN
   ba^ptr ':=' "This is a sample string.";
END;
```

Figure 6

Extended Arrays:

```
<type> .EXT <name> "[" <lower-bound> : <upper-bound> "]"
                        [ := <initialization>] , ... ;
```

where

<type> is

```
{ INT                 }
{ INT(32)             }
{ STRING              }
{ FIXED [ ( <fpoint>) ] }
{ REAL                }
{ REAL(64)            }
```

.EXT is the extended indirection symbol.

<name> is the identifier assigned to the extended array.

<lower-bound> is a 16-bit integer constant defining the first array element. The <lower-bound> must be less than or equal to the <upper-bound>.

<upper-bound> is a 16-bit integer constant defining the last element of the array.

<initialization> is a constant or constant list (including repetition factors) to be assigned as an initial value.

Extended STRUCTS:

```
STRUCT .EXT <name> [ "[" <lower-bound> : <upper-bound> "]" ];
   BEGIN
   < all legal structure data items >
   END;

STRUCT .EXT <name> ( <referral> )
              [ "[" <lower-bound> : <upper-bound> "]" ];
```

where

.EXT is the extended indirection symbol.

<name> is the identifier assigned to the extended structure.

<lower-bound> is the first occurrence of the structure for which storage is allocated.

<upper-bound> is the last occurrence of the structure for which storage is allocated.

<referral> indicates the identifier assigned to a previously defined structure.

Figure 7

```
BLOCK Ext^Global^Example;

  INT .EXT X^Array [0:32767];          ! Double-word pointer is placed in primary
                                       ! memory and array is in extended memory.
                                       ! Array bounds are presently limited
                                       ! to 16-bit integers.

  STRUCT .EXT Y^Struct [0:4095];       ! Double-word pointer is placed in primary
  BEGIN                                ! memory after the X^Array pointer, and it
  INT Y^Array [0:8191];                ! points at the 64-Mbyte array of
  END;                                 ! STRUCTS in extended memory.

  INT Z;                               ! User-memory data can be mixed with
                                       ! extended memory data.
END BLOCK;
```

Figure 7.
Memory mapping for extended global variables.

Issues

Extended Global Variables with Separate Compilation. With the B20 enhancement, users can declare extended arrays and STRUCTS in the same way that user-memory arrays and STRUCTS are declared. Segment allocation and pointer initialization are automatic. Similarly, the bookkeeping for space allocated with separate compilation is handled analogously with that for user-memory data. Extended arrays and STRUCTS can appear in named blocks.

Load on Primary Global Space. A double-word pointer to each extended global variable will be located in the primary storage area. Since the primary global space is 256 words, a maximum of 128 extended global items can be declared.

Figure 7 shows where everything is mapped in memory.

References to extended global variables are not as fast as those in the 16-bit address space, but the available address space can be up to 128 Mbytes (2**27 bytes), whereas regular arrays and STRUCTS are limited by the 64-Kbyte user-data segment size.

Mixing Automatically and Manually Allocated Segments. Users are still free to allocate and initialize their own data segments. However, mixing compiler-allocated and user-allocated segments in a single module is inadvisable. This practice is error-prone because users must remember to explicitly switch among the segments and restore the correct segment before accessing the extended data in that segment.

Dynamic Expansion of the Segment. The extended segment size is fixed at compilation time, and the full segment is allocated at once. Explicit hooks allowing the user to enlarge the segment are not provided.

Local Extended Declarations. Automatic allocation of local extended arrays and STRUCTS will be implemented in the B30 software release. The syntax for that enhancement will be consistent with the syntax of extended global variables.

Unsigned Data Types

Before the B20 release, the scalar data types available in TAL consisted of byte-, word-, doubleword-, and quadword-sized variables. When a programmer needed to define and use bit-oriented variables such as single-bit flags, whose sizes did not correspond to one of the built-in scalar types, there were two possibilities:

1. Use a data type that was large enough to contain the object to be defined, at the expense of some wasted space.
2. Use the bit deposit/extract mechanism provided in TAL to pack two or more bit-variables into the same word of storage and thus minimize wasted space.

Method 1 is fine if programmers can afford the wasted space. Given the limited number of directly addressable locations (global, local, or sublocal) available to a TAL program, however, this is very rarely acceptable. The storage optimization provided by method 2 is usually needed.

Method 2 can be as space-efficient (or inefficient) as programmers care to make it. With suitable DEFINEs, a reasonably convincing simulation of bit-variables could be implemented:

```
INT bit^carrier,
    bit^carrier^;

DEFINE flag0 = bit^carrier.<0>#,
       flag1 = bit^carrier.<1>#,
       flag2 = bit^carrier.<2>#,
       bit3  = bit^carrier.<3:5>#,
       bit5  = bit^carrier.<6:10>#,
       flag3 = bit^carrier.<11>#,
       bit7  = bit^carrier^.<0:6>#;
```

In this example, programmers must first declare the storage needed to contain the bit-variables to be defined and manually allocate each such variable within the declared storage. Manual allocations of this sort are error-prone (it is very easy to inadvertently overlap two bit-variables) and awkward to change. Changing the size of an interior bit-variable, for example, requires manual shifting (reallocation) of one or more adjacent fields.

In the B20 release, TAL implements a new data type, UNSIGNED, which enables programmers to declare bit-variables without having to manually assign bit positions to each variable that is to share the same word of storage. With UNSIGNED types, the preceding example is equivalent to:

```
                          ! Bit Position
UNSIGNED(1) flag0,        !    <0>    (word 0)
            flag1,        !    <1>    (word 0)
            flag2;        !    <2>    (word 0)
UNSIGNED(3) bit3;         !  <3:5>    (word 0)
UNSIGNED(5) bit5;         ! <6:10>    (word 0)
UNSIGNED(1) flag3;        !   <11>    (word 0)
UNSIGNED(7) bit7;         !  <0:6>    (word 1)
```

In this case, the first UNSIGNED variable causes one word of storage to be allocated. Each subsequent bit-variable is then allocated, from bit 0 to bit 15 of this word, in the order it is declared. A bit-variable that is too large to fit causes allocation to begin at bit 0 of a newly allocated word of storage.

Note that because allocation is done by the compiler, changes of size, insertions, and deletions are easily made; the compiler automatically ensures that no overlaps (or unnecessary bit gaps between variables) are introduced.

The syntax for UNSIGNED variable declarations is shown in Figure 8.

Because a bit-variable is not necessarily aligned at a byte or word boundary, it is not "addressable," meaning that it cannot be accessed via a pointer or via a call-by-reference formal parameter in a PROC or SUBPROC.

An UNSIGNED variable can be assigned the value of an expression whose word size is the same as its equivalent INT (or INT(32)) type. Unused bits are discarded from the high-order (bit 0) position of the expression.

Consecutive UNSIGNED variable declarations (excluding formal parameter declarations) as simple variables or as component fields of unpacked structures are allocated, beginning with bit 0 and in the order declared, into contiguous bit positions so that no INT equivalent variable is split across a word boundary and no INT(32) equivalent variable is split across more than one word boundary.

Figure 8

UNSIGNED(<width>) <identifier> [, <identifier>] ... ;

where

<width> is

an integer constant in the range 1 through 31.

UNSIGNED (<width>) defines an arithmetic type whose values are unsigned integers having binary representations that are <width> bits wide. This type can be used to declare global, local, and sublocal variables, call-by-value formal parameters in a PROC or SUBPROC, and component fields within STRUCTs and STRUCT templates.

The value of an UNSIGNED type is logically the same as type INT if <width> is 15 or less and INT(32) otherwise. Consequently, an UNSIGNED type has the same operations (e.g, -, +, *, /, and '-') as its equivalent INT (or INT(32)) type.

Figure 8.
Syntax for UNSIGNED variable declarations.

Currently, arrays of UNSIGNED variables cannot be declared and UNSIGNED variables cannot be subscripted. These features will be supported in a future release.

Within structures, the allocation of UNSIGNED variables is very similar to their allocation outside structures:

```
STRUCT s;
 BEGIN                    ! Bit Position
 UNSIGNED(1) flag0,       !    <0>    (word 0)
             flag1,       !    <1>    (word 0)
             flag2;       !    <2>    (word 0)
 UNSIGNED(3) bit3;        !  <3:5>    (word 0)
 UNSIGNED(5) bit5;        ! <6:10>    (word 0)
 UNSIGNED(1) flag3;       !   <11>    (word 0)
 UNSIGNED(7) bit7;        !  <0:6>    (word 1)
 END;
```

Because UNSIGNED variables can be declared to be wider than 16 bits, TAL can support accesses to bit-variables that are split across word boundaries. Previously, this could not be simulated with bit deposits/extracts because those operations are restricted to 16-bit expressions. This example shows the new feature:

```
                          ! Bit Position
UNSIGNED(1)  flag0,       !    <0>    (word 0)
UNSIGNED(23) bit23;       !  <1:15>   (word 0)
                          !  <0:7>    (word 1)
UNSIGNED(5)  bit5;        !  <8:12>   (word 1)
UNSIGNED(1)  flag3;       !   <13>    (word 1)
```

Finally, two new standard functions similar to the existing $LEN and $OFFSET functions have been implemented. The new functions are $BITLENGTH(<variable>) and $BITOFFSET(<variable>). For any variable, $BITLENGTH returns the minimum number of bits allocated for that variable. For any variable within a structure, $BITOFFSET returns the bit offset of that variable within the outermost containing structures. In the two preceding examples, $BITLENGTH(bit23) returns the value 23 and $BITOFFSET(s.bit7) returns the value 16.

Literal Declarations

The use of symbolic literals can greatly improve the readability of a TAL program. In many cases, the existing form of literal declaration in TAL is more than sufficient. Often, however, a programmer may want to define a set of logically related, ordered symbolic values using integer literals. Here, the basic intent is that each symbolic name is to have a unique integer value; the value of a particular symbolic literal may or may not be significant.

Figure 9.
Syntax for literal declarations.

Figure 9

```
LITERAL <literal definition>
        [ , <literal definition> ] ... ;
where
<literal definition> is
    { <explicit literal> }
    { <implicit literal> }

<explicit literal> is
    <identifier> = <constant>

<implicit literal> is
    <identifier>

<identifier>
    is any legal TAL identifier.

<constant>
    is an INT, INT(32) constant expression, FIXED,
    REAL, or REAL(64) constant.
```

Before the B20 release, a natural way to ensure uniqueness was to define the symbolic literals as an increasing (incrementing by one) sequence of integers. There were two ways to proceed.

One was to assign integer values explicitly:

```
LITERAL
    dollar^aus = 0,
    dollar^can = 1,
    krone^den = 2,
    franc^fr   = 3,
    dollar^hk  = 4,
    yen        = 5,
    dollar^usa = 6,
    pound^uk   = 7,
    mark^frg   = 8;
```

The second way was to assign each succeeding literal a value that is one greater than its predecessor:

```
LITERAL
    dollar^aus = 0,                 ! = 0 !
    dollar^can = dollar^aus + 1,    ! = 1 !
    krone^den = dollar^can + 1,     ! = 2 !
    franc^fr   = krone^den + 1,     ! = 3 !
    dollar^hk  = franc^fr + 1,      ! = 4 !
    yen        = dollar^hk + 1,     ! = 5 !
    dollar^usa = yen + 1,           ! = 6 !
    pound^uk   = dollar^usa + 1,    ! = 7 !
    mark^frg   = pound^uk + 1;      ! = 8 !
```

Both methods are error-prone. With either method, the programmer must ensure that two literals are not duplicated. Also, insertions and deletions require that one or more adjoining literals be changed as well.

In the B20 release, TAL extends the syntax for literal declarations to permit a simpler, easier alternative:

```
LITERAL dollar^aus,  ! = 0 !
        dollar^can,  ! = 1 !
        krone^den,   ! = 2 !
        franc^fr,    ! = 3 !
        dollar^hk,   ! = 4 !
        yen,         ! = 5 !
        dollar^usa,  ! = 6 !
        pound^uk,    ! = 7 !
        mark^frg;    ! = 8 !
```

Here, the TAL compiler initializes the sequence to 0 and automatically increments successive values by one, ensuring that duplications do not occur. Insertions and deletions can be made without having to change any other literal in the list.

Figure 9 shows the new syntax for literal declarations.

Literal declarations using only <explicit literal> have the same meaning as in earlier releases.

<Implicit literal>, when used, must be either the head of a list of <literal definition>s or it must be preceded by a <literal definition> (implicit or explicit) that defines an INT(16) constant literal. Under the former condition, the value of the literal is set to 0; under the latter, the value of the literal is set to one greater than the value of its predecessor in the same literal declaration.

The following examples show how this extension is used:

```
LITERAL zero, ! = 0 !
        one,  ! = 1 !
        two;  ! = 2 !

LITERAL not^3;
  ! Equals 0, not 3. !

LITERAL three = 3;

LITERAL not^4;
  ! Equals 0, not 4. !

LITERAL five = 5, ! = 5 !
        six,      ! = 6 !
        seven;    ! = 7 !

LITERAL pi = 3.141592e0,
        pi^plus^1;
        ! ILLEGAL, predecessor is
        ! not an INT(16) value.

LITERAL int^32 = 1d,
        int^32^plus^1;
        ! ILLEGAL, predecessor is
        ! not an INT(16) value.

LITERAL root^2 = 1.414e0;

LITERAL implicit^zero;     ! = 0 !
```

MOVE Statement

The B20 release enhances the TAL MOVE statement in two ways:

1. Users are now allowed to mix 16-bit byte-addressed destinations with 16-bit word-addressed sources, and vice versa.
2. A count unit descriptor has been introduced to make explicit the size of the data elements being moved.

Figure 10

```
<destination> { ':=' } { <source> FOR <count> [<count unit>] }
              { '=:' } { <constant>                          }
                   [ & <source> FOR <count> [<count unit>] ]...
                   [ & <constant>                          ]...
                                  [ -> <next address> ]
```

<destination>
is the name of the variable, with or without an index, to which the move begins. It can be a simple variable, array, pointer, structure, substructure, structure data item, or structure pointer, but not a read-only array.

':='
indicates a left-to-right sequential move.

'=:'
indicates a right-to-left sequential move.

<source>
is the name of the variable, with or without an index, from which the move begins. It can be a simple variable, array, read-only array, pointer, structure, substructure, structure item, or structure pointer.

<count>
is a positive INT arithmetic expression that defines the number of bytes, words, or elements in <source> to move.

<count unit>
is a description of <count>. It is one of
BYTES
WORDS
ELEMENTS
If <count unit> is not present, the meaning of <count> is as follows:

<source>	<count> is the number of
Simple variable	elements
Array	elements
Structure	words
Substructure	bytes
Structure pointer	bytes if STRING, words if INT
Pointer	elements

<constant>
is the LITERAL, numeric or character string constant, or constant list to be moved.

<next address>
is a variable to contain the location in <destination> that follows the last item moved. <next address> is a 32-bit byte address if either <source> or <destination> has an extended address; a 16-bit byte address if both <source> and <destination> have standard byte addresses; or a 16-bit word address if both <source> and <destination> have standard word addresses.

Figure 10.
Syntax description of the MOVE statement.

The addition of these two features increases the usefulness of the MOVE statement and improves its readability. Also, more efficient code is now generated for MOVE statements that mix standard addresses and extended addresses.

The new MOVE statement syntax is upward-compatible with the previous syntax. (See Figure 10 for a formal syntax description.)

Byte and Word Addressing

Before the B20 release, users were not allowed to mix standard byte and word addressing in the MOVE statement. For example, the following would generate an error:

```
PROC p;
  BEGIN
    STRING s[0:9];
    INT   i[0:4];
    s ': = ' i FOR 5;
          ^
    **** ERROR 32 **** Type incompat-
    ibility ** I
  END;
```

The error message was emitted because the user had a byte-addressed <destination> and a word-addressed <source>.

In the B20 release, the TAL compiler recognizes 16-bit byte/word mismatches in the MOVE statement and emits an extended move sequence. In such cases, TAL generates the extended address of both the <destination> and <source> variable, generates a byte count (equivalent to <count>), and emits the extended hardware instruction to accomplish the move. Whenever TAL generates an extended move sequence the <next address> variable, if specified, must be a 32-bit variable; otherwise, TAL emits an error message.

Two programming considerations should be noted. First, if the <destination> is word-addressed and the <source> is byte-addressed, the <next address> variable will not point to an element boundary if an odd number of bytes are moved. For example:

```
PROC p;
  BEGIN
    STRING s[0:9];
    INT      i[0:4];
    STRING .EXT s^next^addr;
    i ': = ' s FOR 3 -> @s^next^addr;
  END;
```

When the move is complete, s^next^addr points to the right-hand byte of i[1] (bits <8:15> of i[1]).

The second consideration is that this feature of TAL is only available if the resulting object file will be run on a NonStop system. (Since TAL is generating an extended hardware instruction, the object file cannot run on a NonStop 1+™ system.) TAL emits an error 32 (type incompatibility) if the target system for the compilation is a NonStop 1+ system and users attempt to mix 16-bit byte- and word-addressed variables in a MOVE statement.

Count Unit

Before the B20 release, the unit size of the elements moved was not specified explicitly in the syntax, but rather was dependent on the type of the source variable. Thus, in order to understand what a particular MOVE statement actually did, a programmer reading an unfamiliar piece of TAL source code would have had to refer back to the declarations of the source and destination variables.

In the B20 release, the count unit has been added, so that programmers can specify whether to move in units of BYTES, WORDS, or ELEMENTS. The meaning of ELEMENTS depends on the type of the source variable as listed below:

<Source>type	Size of ELEMENTS
STRING	Byte
INT	Word
INT(32)	Doubleword
FIXED	Quadword
REAL(32)	Doubleword
REAL(64)	Quadword
STRUCT	One occurrence of the struct
STRING STRUCT POINTER	One occurrence of the struct
INT STRUCT POINTER	One occurrence of the struct
SUBSTRUCT	One occurrence of the substruct

If source is a STRUCT or STRUCT pointer and the <count unit> ELEMENTS is used, then an even number of bytes is always moved. In other words, when $LEN is odd, the pad byte at the end of the structure is also moved.

If an identifier other than BYTES, WORDS, or ELEMENTS is used as a <count unit>, TAL emits an error message.

Figure 11a shows equivalent forms of the same MOVE statement that use different <count unit>s. Figure 11b shows how the

BYTES <count unit> can be used to override the MOVE statement units when both the <source> and <destination> are type INT variables.

Note that the identifiers BYTES, WORDS, and ELEMENTS used as a <count unit> are not TAL reserved words, so they can be used as variable names. There is, however, one restriction. Identifiers used as a <count unit> in a MOVE statement cannot also be used as names of a DEFINE or a LITERAL. This is illustrated in the following example:

```
PROC example^5;
 BEGIN
  INT var1[0:11];
  INT var2[0:11];
  LITERAL bytes = 12;
  DEFINE words = var1 + var2 #;
  var1 ': = ' var2 FOR 24 BYTES;
                          ^
  **** Error 27 **** Illegal Syntax
  var1 ': = ' var2 FOR 12 WORDS;
                          ^
  **** Error 27 **** Illegal Syntax
  END;
```

ERRORFILE Directive

Before the B20 release, users did not have an easy way to log compilation errors or warnings and then edit their source code while viewing the errors.

In the B20 release, a new TAL directive called ERRORFILE logs any compilation errors or warnings to a specified file. Then after compilation, users can edit their source while viewing the errors in Tandem's new editor, PS TEXT EDIT™ (TEDIT).

Users can log compilation errors or warnings to a file by entering the ERRORFILE directive on the run line as in

```
TAL /IN MYSOURCE, OUT $S/ MYOBJ;
    ERRORFILE <file name>
```

or in the source code before any declarations, as in

```
?ERRORFILE <file name>
  INT MY^FIRST^DEC;
```

TAL will then create an entry-sequenced file called <file name> with a file code of 106. <File name> will be expanded with the name of the current default volume and subvolume if necessary.

Figure 11

(a)

```
PROC example^1;
  BEGIN
  INT count;
  STRING .s^buf^one [0:79];
  STRING .s^buf^two [0:79];
  INT    .i^buf^one [0:39];
  INT    .i^buf^two [0:39];
  INT(32) .d^buf^one [0:19];
  INT(32) .d^buf^two [0:19];

! Each of the following statements is equivalent:
s^buf^one  ': = '  s^buf^two FOR 80;
s^buf^one  ': = '  s^buf^two FOR 80 BYTES;
s^buf^one  ': = '  s^buf^two FOR 80 ELEMENTS;

! Each of the following statements is equivalent:
count : = 40;
i^buf^one  ': = '  i^buf^two FOR count;
i^buf^one  ': = '  i^buf^two FOR count WORDS;
i^buf^one  ': = '  i^buf^two FOR count ELEMENTS;

! Each of the following statements is equivalent:
d^buf^one  ': = '  d^buf^two FOR 20;
d^buf^one  ': = '  d^buf^two FOR 80 BYTES;
d^buf^one  ': = '  d^buf^two FOR 40 WORDS;
d^buf^one  ': = '  d^buf^two FOR 20 ELEMENTS;

END; ! example^1
```

(b)

```
INT .int^global^buf [0:66];
PROC example^2;
  BEGIN
  INT file^number;
  INT .int^local^buf[0:66];
  INT count^read;
  CALL READ(file^number, int^local^buf, 132, count^read);
  ! If we wish to copy this record into int^global^buf using the byte (not word)
  ! count returned in "count^read":
  int^global^buf ': = ' int^local^buf FOR count^read BYTES;
  ! This is equivalent to the following "word-oriented" form:
  int^global^buf ': = ' int^local^buf FOR (count^read + 1) / 2;
  END; ! example^2
```

TAL writes one record to <file name> for each error or warning that occurs during the compilation. Each record has this information in the following order:

1. The external form of the source file that contains the source line in which the error or warning occurred.
2. The edit line number of the line in which the error or warning occurred.
3. The column number in the line where TAL detected the problem.
4. The error or warning message text.

Figure 11.
MOVE statement count units. (a) Equivalent moves using different count units. (b) Overriding the source type with a count unit.

Figure 12

```
TAL /IN MYSOURCE, OUT $S.#HOLD/ MYOBJ; ERRORFILE myerrs
                        ? ERRORFILE myerrs
    1.    000000 0  0     int i;
    2.    000000 0  0     string s := 1000;
                                      ^
**** WARNING **** 13 — Value out of range
    3.    000000 0  0     proc p;
    4.    000000 1  0     begin
    5.    000000 1  1
    6.    000000 1  1       CALL s;
                              ^
**** ERROR **** 31 — Only PROC or SUBPROC identifier allowed ** S
    7.    000000 1  1       i := j;
                               ^
**** ERROR **** 49 — Undeclared identifier ** J
                                ^
**** ERROR **** 32 — Type incompatibility ** I
    8.    000003 1  1
    9.    000003 1  1     end;

  Errorfile file name is $TAL.TL0123.MYERRS
  Number of compiler errors = 3
  Number of compiler warnings = 1
  Maximum symbol table space used was = 16898 bytes
  Number of source lines = 10
  Compile time - 00:00:44
  Total elapsed time - 00:00:46

The error file, myerrs, will show:
  $TAL.TL0123.MYSOURCE  2.  40 **** WARNING **** 13 — Value out of range
  $TAL.TL0123.MYSOURCE  6.  32 **** ERROR **** 31 — Only PROC or SUBPROC
                                                     identifier allowed ** S
  $TAL.TL0123.MYSOURCE  7.  31 **** ERROR **** 49 — Undeclared identifier ** J
  $TAL.TL0123.MYSOURCE  7.  32 **** ERROR **** 32 — Type incompatibility ** I
```

Figure 12.
An example of the ?ERRORFILE directive used to direct error messages to a file.

At the end of the compilation, TAL prints the name of the ERRORFILE in the trailer message. Figure 12 shows a partial listing.

Users wishing to view both the source file and the error file in the editor should proceed as follows. First, transfer the data in the error file to an edit file because TAL writes an entry-sequenced file. For example:

```
EDIT myerrs p emyerrs; exit
```

The "e" prefix denotes that this is the edit form of the error file.

Next, enter TEDIT and use both windows to edit the source file, MYSOURCE, and inspect the error file, EMYERRS:

```
TEDIT mysource; OPENWINDOW 2,emyerrs;
SWITCHWINDOW
```

(If not enough of the message text is visible in EMYERRS, switch windows to see more of the text by using the TEDIT command, LEFTSCROLL.)

Users can create the edit form of the error file and enter TEDIT simultaneously with the following macro:

```
?section talerror macro
edit %2% p e%2% !; exit
tedit %1%; op 2,e%2%; sw
```

If this macro is put in a file called TALMAC, the TACL command LOAD TALMAC will make the macro name, TALERROR, known to TACL™ (the Tandem Advanced Command Language). To simultaneously transfer the data in the error file, MYERRS, to the edit file, EMYERRS, and enter TEDIT, enter the new TACL command TALERROR MYSOURCE MYERRS.

Note that when TAL is processing the ERRORFILE directive:

- If <file name> does not exist, TAL creates it as specified.
- If <file name> exists and its file code is 106, TAL purges <file name> and then creates <file name>.
- If <file name> exists and its file code is not 106, TAL terminates the compilation.

Acknowledgment
The authors would like to acknowledge the substantial contributions made by Jeff Lichtman, both in the implementation of many of the additions to TAL described in this article and in the preparation of this article. Jeff served as technical leader for B20 TAL development and was responsible for designing and implementing the MOVE statement enhancements, the structure enhancements, and the ?ERRORFILE compiler directive.

Catherine Lu is a software developer working on the TAL compiler. She joined Tandem in September 1984 after completing a B.S. in Computer Science and Engineering at Massachusetts Institute of Technology.

John Murayama has worked on the TAL compiler since joining Tandem in July 1984. For the B20 release, he designed and implemented the labeled CASE statement and the UNSIGNED data type. Currently, he is technical leader of the TAL compiler project. Before joining Tandem, he developed language processors and software development tools for other computer vendors.

Predicting Response Time in On-line Transaction Processing Systems

This article describes a method of predicting response time in balanced transaction processing applications that use Tandem's PATHWAY transaction processing system. Its intent is to show how the basic elements of contention in a Tandem NonStop system can be modeled with standard modeling techniques. The method described will not help readers tune their systems for better performance; rather, it will allow them to predict the performance obtainable from their systems if the systems are well tuned.[1]

Overview

The PATHWAY transaction processing system facilitates the design of transaction processing applications through the implementation of requester-server structures. The requester function is implemented by the PATHWAY Terminal Control Process (TCP), which manages terminal I/O. The TCP accepts transaction requests from the terminals and passes them to servers. The servers provide data-base service by making appropriate requests to the disk processes and then replying to the TCP. Figure 1 illustrates a typical application using the PATHWAY system.

The performance of an application that uses PATHWAY can be characterized by the amount of time the system takes to respond to transaction processing requests. This time is known as *response time*.

Response time can be expressed as a sum of service time and queuing time. The *service time* of a transaction is the time required by the system to process the transaction when only one transaction is active in the system. When multiple transactions are active, some of them may have to wait for the processors, disks, and other system resources to become available.

Figure 1

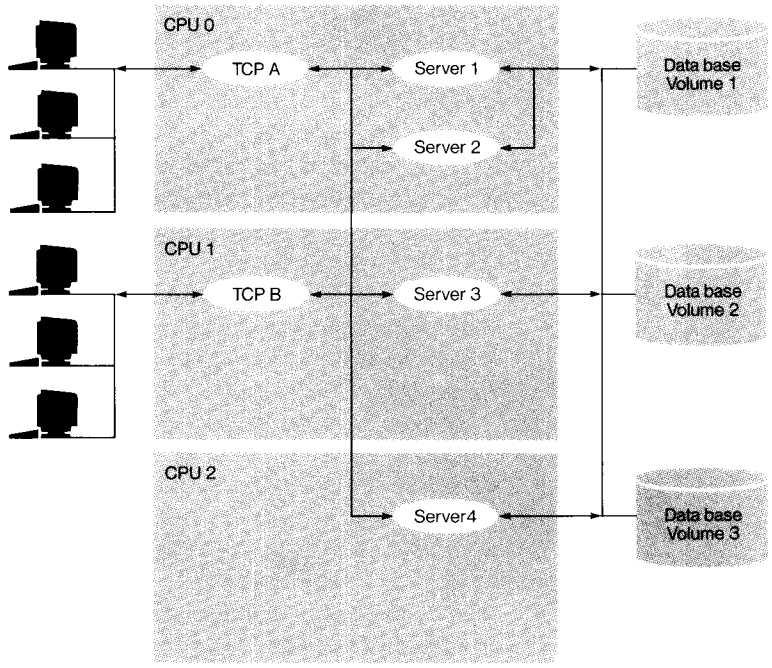


Figure 1.
Overview of an application system that uses the PATHWAY transaction processing system.

[1]Note that response-time prediction on Tandem NonStop systems requires a great deal of knowledge about Tandem hardware and software, along with expertise in queuing network modeling and event-driven simulators. The method presented here is not a "quick fix" to the problem of response-time prediction.

The delay incurred by this wait is called the *queuing time*. Queuing time is a function of the transaction rate; hence, response time is also a function of the load on the system. Characterizing the performance of an application that uses the PATHWAY system involves determining the response time of the system as a function of the transaction rate.

The Model

For most balanced applications using the PATHWAY system, service time varies only slightly with the processing load. Although the service time for those applications that use the Transaction Monitoring Facility (TMF) and operate at very high loads does vary appreciably with the load, the variation in other applications that use PATHWAY is usually less than 5%.

Hence, for a response-time prediction model, it is reasonable to assume that service time is independent of processing load. Given this assumption, one can calculate response time if the queuing delays in the system can be determined as a function of the load.

The following describes a simulation model that can be used to find these queuing delays. This model is an extremely simplified version of the response-time prediction model used in ENVISION. Tandem's system analysts use ENVISION, a performance modeling tool, to size, tune, and predict the performance of online transaction processing systems (Chou, Oleinick, and Singh, 1984).

The model simulates the basic elements of contention in Tandem NonStop systems. It is expected to give reasonable results for systems using Disc Process 2 (DP2). Systems using Disc Process 1 (DP1) have additional elements of contention that require different modeling techniques.

As mentioned earlier, the reader is assumed to be familiar with basic queuing theory (Kobayashi, 1978), event-driven simulators (MacDougall, 1980), and Tandem hardware and software.

Modeling Method

The system is modeled as a network of queues. In this type of model, all system entities, such as processors, disks, and terminals, are represented as *resources*, and tasks are modeled as *jobs*.

Resources can be active or passive. An *active resource* is a facility providing some service, along with a "waiting room" or queue. A *passive resource* does not provide service, but is needed to accomplish a job; e.g., a thread of the DP2 disk process is required before the job can be served at the processor. Jobs requesting service at a busy resource wait in the queue for that resource. All the jobs in the queue are served in a prespecified manner.

After obtaining the required service from a resource, jobs are routed to other resources. The set of resources in the system forms a network of queues. An explanation of how the jobs are routed within this network, along with the specification of the entities in the network, describes the system completely.

Assumptions

A typical Tandem NonStop system running the PATHWAY transaction processing system consists of several processors and disks. There are several requesters and servers, and a multithreaded disk process for each disk volume. The system also has other processes, such as line handlers, a monitor, and a memory manager. In systems using TMF, every processor has a TMF monitor process, and one TMP process in the system manages TMF audit trails and helps with the processing of the TMF transactions that modify data over more than one node in a network.

Although the processing done by each of these processes for the transaction can be represented explicitly for a given system, describing the topology and the routing logic for a general case is a formidable problem. Moreover, for systems being sized for future applications, some of this information is unclear or

not available. The model described here uses an assumption to reduce this complexity to manageable levels, and has been shown to yield reasonably accurate results.

The model assumes that the processing load is balanced. This implies that all the CPUs are more or less equally utilized. With this assumption, it is possible to model the behavior of the multiprocessor system with a single CPU and a disk subsystem. The assumption is not as restrictive as it might appear. The model has been found to work well when processor utilizations are within 10% of each other.

The model also assumes sufficient replication of the servers. Since servers written for the PATHWAY transaction processing system are single threaded, this assumption means that no transaction waits for a server to become available. Although this assumption is not valid if the system is operated at high transaction rates with few servers, note that users can use PATHWAY to change the number of servers to get better performance at high loads. Thus, this assumption is valid for a well-tuned system.

Model Description

The system model consists of a processor, a terminal cluster, and a disk subsystem, as shown in Figure 2. Although the model represents a greatly simplified view of a typical system that comprises multiple processors and disks, it works because it assumes a balanced load.

Resources. The processor is modeled as a priority queue having different priorities for the disk process, servers, TCPs, line handlers, and interrupts. The relative priority of these processes are:

Process	Priority
Interrupts	3 (highest)
Disk process	2
Line handler	1
TCP and servers	0 (lowest)

The processor queue is also a preemptive queue. Thus, the arrival of a new job having a priority higher than the currently executing job causes the execution of the current job to be suspended, allowing the execution of the job that has just arrived. Within each priority level, all jobs are served on a first-come, first-served basis.

Figure 2

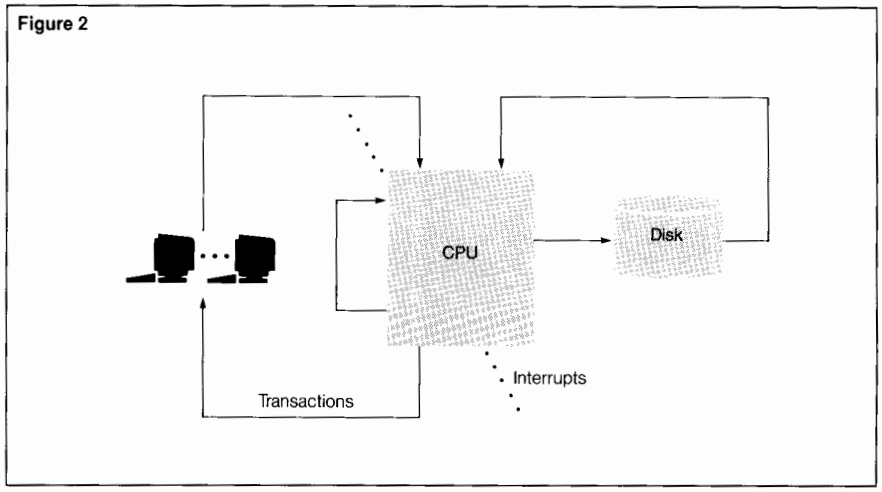


Figure 2.
The model. Open-class jobs are represented by dotted lines, closed-class jobs by solid lines.

The terminal cluster is modeled as an "infinite server" station; i.e., it does not cause the jobs to queue up, but delays them by a certain amount of time. This reflects *think time*, during which the users "think" for some time after a transaction completes, before starting the next transaction. (For a more detailed discussion of think time, see Kosinski, 1984.)

The disks are modeled as a multiple-server facility, serving jobs with a first-come, first-served discipline. The number of servers in the facility is a function of the ratio of processors to disks. For example, if the system were to have four processors and eight disks, the disk subsystem facility would have two servers. In reality, a different queue exists for each disk volume, the ratio of processors to disks need not be an integer, and contention for a disk volume also depends on the relative proportion of read and write requests made to that disk.

Although the model is a crude representation of the disk subsystem, it does not introduce major errors in response-time prediction because systems using DP2 have fewer physical I/Os than logical I/Os. This is because DP2 uses buffered cache rather than write-through cache to reduce the number of disk I/Os. Hence, even at very high system loads, disk queuing times are minimal.

Figure 3

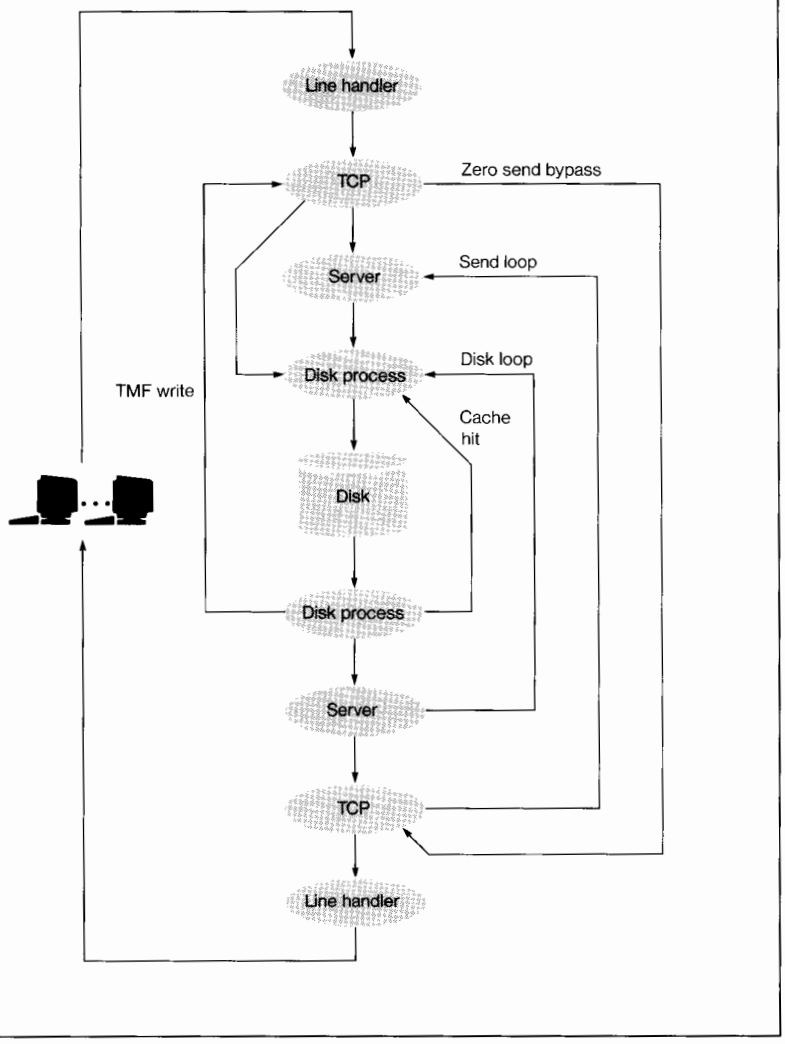


Figure 3.
Routing logic for closed-class jobs used in the network shown in Figure 2.

Jobs. Two types of job circulate in the queuing network described above: transactions and interrupts. Transactions are modeled as *closed-class* jobs, i.e., jobs that always remain in the queuing network. This modeling mechanism is used to reflect the fact that the maximum number of active transactions in the system equals the number of terminals.

The transactions originate at the terminals and are routed to the processor for line-handler processing. These jobs then circulate between the processor and the disk, depending upon the characteristics of the transaction. After receiving the required service from the CPU and the disk, the job returns to the terminals. There, another transaction originates at a future time, depending on the think time of the terminal.

Interrupts are modeled as *open-class* jobs, i.e., jobs generated by a random source. The system has a variable number of open-class jobs at any time, while the number of closed-class jobs remains static. The open-class jobs are generated in a random fashion, served by the processor, and then exit the system forever.

Open-class jobs model increased processor contention caused by parallelism inherent in Tandem NonStop systems. They model the fact that more than one processor can do the processing for a transaction at the same time. The amount of CPU time used by a transaction is about 25% more than the service time at the CPU. The difference, modeled by open-class jobs, causes greater CPU contention, although it is not a part of the service time.

Routing. Routing of closed-class jobs between the processor and disk is determined by the characteristics of the transaction. Two factors affect routing: the number of sends done by a transaction and the number of data-base accesses during each send. (Note that a send is a single message with a reply between a requester and a server.) For audited transactions, the number of TMF audit-trail writes also affects routing.

Figure 3 illustrates the generalized routing pattern for a transaction. From the terminal cluster, the transaction is routed to the processor for line-handler processing and then TCP processing. There are two processing loops, the send loop, executed once for every send, and the disk loop, executed once for every data-base access request.

The *disk loop* is initiated when the server makes a data-base access request. The disk-process code is executed at the processor, and the job is routed to the disk in the event of a cache miss. After returning from the disk, the job undergoes more disk-process processing. The disk process then replies to the server, completing the disk loop.

In the *send loop*, TCP processing for every send is done at the processor, and then the job is routed to the processor again for server processing. The server calls the disk process for every data-base access required by that

send, initiating a disk loop. After the execution of every disk loop, control returns to the server, and server code is executed at the processor. When the required number of disk loops have been executed, the server replies to the TCP, which may then initiate another send loop if the transaction does more than one send.

After the required number of send-loop executions, the job receives line-handler processing and returns to the terminal. This finishes the processing for the transaction. Another transaction originates at the terminal after think time.

For transactions protected by TMF, additional visits to the processor are made for disk-process processing. When the TCP processes ENDTRANSACTION, it sends a message to the TMF Monitor process, which initiates the processing that causes disk processes to write their audit buffers to disk.

Since the processing time required by the TMF Monitor process is small, ENDTRANSACTION processing is modeled by routing the job from the TCP to the disk process for every audit-trail write. After the disk process is served by the processor, the job is routed to disk. When disk processing is complete, the job is routed to the processor for post-processing by the disk process, and then to the TCP. This loop is repeated for every audit-trail write.

Model Parameters

Response-time prediction based on the model described above requires values for the number of processors, disks, and terminals, along with the following pieces of information about each transaction.

Transaction Rate. The *transaction rate* is the number of transactions per second. The system is modeled by one representative processor; hence, the transaction rate used for each transaction should be the transaction rate per processor.

Number of Sends. One can determine this number by examining the SCREEN COBOL code interpreted by the TCP. It is the number of SENDs per transaction.

Number of Disk-process Requests per Send. This is the number of data-base access (logical I/O) requests made by the server for each send. Note that there may be a different number of disk-process requests for each send.

CPU Processing Time for the Transaction. This time comprises the following:

- Line-handler time.
- TCP time.
- Server time.
- Disk-process time.
- Miscellaneous time, plus interrupts.

If the application is currently running, these times can be measured with the XRAY™ performance analysis tool; otherwise, the process of collecting this information is more involved.

Server and disk-process times can be estimated by summing up the CPU times for the File System operations (such as reads, writes, and updates) done by the server. One can determine the CPU time required by the server and the primary and backup disk processes for File System operations by running simple test scripts for each operation and examining the XRAY results. Although this can be tedious, note that once determined, these atoms can be used for all other transactions involving the same operations.

The CPU time used by a transaction is about 25% longer than the service time at the CPU.

Similarly, *TCP time* can be estimated by summing up the CPU times of all its components. These components are: (1) the time required by the primary TCP to do a send, checkpoint, BEGINTRANSACTION, ENDTRANSACTION, and so on; and (2) the time required by the backup TCP to process checkpoints.

Disk-service Time for Each Transaction. This time can be reported by the XRAY DISC BUSY counter if the application is currently running. If the application uses unmirrored disks, the service time is simply the DISC BUSY time per transaction; otherwise, the service time can be

approximated by adding the primary disk's DISC BUSY counter, the mirror's READ BUSY counter, and the read proportion of the mirror's SEEK BUSY counter. This computation is required because the data can be read from either the primary or the mirror disks, whereas data must be written to both the primary and the mirror.

If the application is not currently running, one must build disk-atom tables for each File System operation and estimate the DISC BUSY time from them.

Audit-trail Writes. For all audited transactions, it is necessary to estimate the number of writes to the audit trail. Although the actual number of audit-trail writes for a given application is a function of several factors, one can obtain a reasonable approximation using the method outlined below. This method produces a conservative estimate of the number of audit-trail writes; the actual number is usually lower.

In order to estimate the number of audit-trail writes, one must compute the amount of audit data generated by the transaction. The number of audit bytes can be computed by summing the audit data generated by each File System operation done by that transaction. Assuming audit-trail compression to be off, n_{audit_bytes}, the number of audit bytes generated by each operation modifying a record in an audited file, is

$$n_{audit\_bytes} = 62 + l_{before\_image} + l_{after\_image} .$$

Here, l_{before_image} and l_{after_image} are equal to the record size if the "before" and "after" images for the operation exist. For writes, no before image exists and l_{before_image} is 0; for deletes, no after image exists and l_{after_image} is 0; for updates, both before and after images exist. When audit-trail compression is on, the number of audit bytes depends on the difference between the before and after images. For updates, it is impossible to compute this value; one can only obtain it by running an application benchmark.

Once the number of audit-trail bytes for each transaction is known, the number of audit-trail writes depends on whether there are one or two audit trails. When the system has only one, called the master audit trail (MAT), it is easy to estimate the number of audit-trail writes. If a transaction generates x bytes of audit data, n_{MAT_writes}, the number of MAT writes for that transaction can be approximated by

$$n_{MAT\_writes} = \left(1 + \frac{x}{audit\_block\_size}\right).$$

The number of MAT writes can be approximated in this way because, for every transaction, the audit buffer is written once to the disk, accounting for the first term on the right-hand side. In addition to this write, occasionally the audit buffer spans block boundaries; i.e., the audit data is in two physical blocks. This causes an additional write to the audit trail, accounted for by the second term. For DP2, the default value of *audit_block_size* is 4 Kbytes.

When a system uses multiple audit trails, the audit-trail writes to the MAT can be computed as shown above. Computation of writes to the auxiliary audit trails (AATs) is slightly more involved. A typical audit trail receives audit data from several disk processes. Since the operations on each file and the disk where each file is located are known, the amount of audit data from each disk can be computed. (This computation is similar to the one shown above.)

If the amount of audit data from the i^{th} disk process is x_i, and n disk processes send their audit data to the auxiliary audit trail m (AAT_m) under consideration, then the number of audit-trail writes is the sum of audit-trail writes by all the disk processes involved, i.e.,

$$num\_writes_{AAT_m} = \sum_{i=1}^{n} \left(1 + \frac{x_i}{audit\_block\_size}\right).$$

The number of audit-trail writes affects the disk-process time (part of the CPU processing time, above) and the disk-service time. Given the disk-process and disk-service time for a disk write of *audit_block_size*, one can approximate the additional time because of the audit-trail writes.

Implementation

The queuing network described above can be simulated quite easily with a simulation language, such as Simula. If such a language is not available, one can use any high-level language to write an event-driven simulator. Since the network consists of only three resources, writing the simulator is relatively easy.

Think-time Estimation

Response-time prediction for a given transaction rate requires an estimate of the average think time at the terminals. One can obtain this average as follows.

The transaction rate per terminal allows one to determine the intertransaction time at the terminals: it is simply the inverse of the transaction rate, or the sum of the average think time and the average response time. Hence, estimation of think time requires an estimate of the average response time, as described below.

Given the transaction rate and the processor and disk times for each transaction, one can determine the processor and disk utilizations. These can be used to compute an initial estimate of the response time using an approximation of *M/M/1*. If the system has more than one type of transaction, a weighted average of response times should be used.

Subtracting the response time, obtained above, from the intertransaction time gives an initial estimate of think time. A short simulation run with this time yields a better estimate of the response time, hence, the think time at the given transaction rate. Another run of the simulation model with the new value of think time yields a still better estimate of response time. This iterative process can be continued to yield an acceptably close value for think time. Then, a simulation run of sufficiently long duration will yield the response times at the given transaction rate.

Duration of the Simulation Run

Average response times are obtained by finding the average of the response-time values observed during the simulation run. The duration of the simulation run determines the statistical spread of the response-time values for each transaction. The longer the simulation duration, the smaller the spread. This statistical variation in the values is a function of several variables, e.g., the routing patterns of the transactions, the transaction rates, and the processor and disk utilizations.

Although it is not possible to predict the variation in values for a given application, observations show that a simulation run yielding approximately 1000 data points is sufficient to reduce the error in the average response time of the system to within 5%. Similarly, if a response time with a statistical error of less than 5% is required for a specific transaction, the simulator should be run long enough to collect approximately 1000 data points for that transaction.

Benchmark Results

In this section, predicted response times produced by the model for a large banking application are compared with measured response times for that application.

The application benchmark was run on a four-processor NonStop TXP system having four mirrored disk volumes. NonStop TCPs were used, and all application files were audited by TMF. The transaction flow is outlined below, and the data-base files are described in Table 1.

TCP flow

Accept 100 bytes.
Begin transaction.
Send to server.
End transaction.
Perform 10 IF statements.
Perform 10 MOVE statements.
Perform 5 ADD statements.
Perform 5 SUBTRACT statements.
Display 200 bytes.

Server flow

Read 100 bytes from TCP.
Read Account file.
Read Teller file.
Read Branch file.
Update Account file.
Update Teller file.
Update Branch file.
Write History record.
Perform 10 IF statements.
Perform 10 MOVE statements.
Perform 5 ADD statements.
Perform 5 SUBTRACT statements.

For this benchmark, Figure 4 charts the measured and predicted values of the average response time as a function of the transaction rate. Tandem's internal performance-modeling tool, ENVISION, from which the simplified model presented in the article was derived, was used to predict the values of the atoms of CPU and disk contention.

Note that the response times predicted by ENVISION are nearly identical to the measured values. The simplified model described in this article is expected to predict the response times to within 20% of actual values, except at very high loads.

Figure 4.
Measured versus predicted response times for the application benchmark. Note that the predicted response times are well within 20% of the measured values.

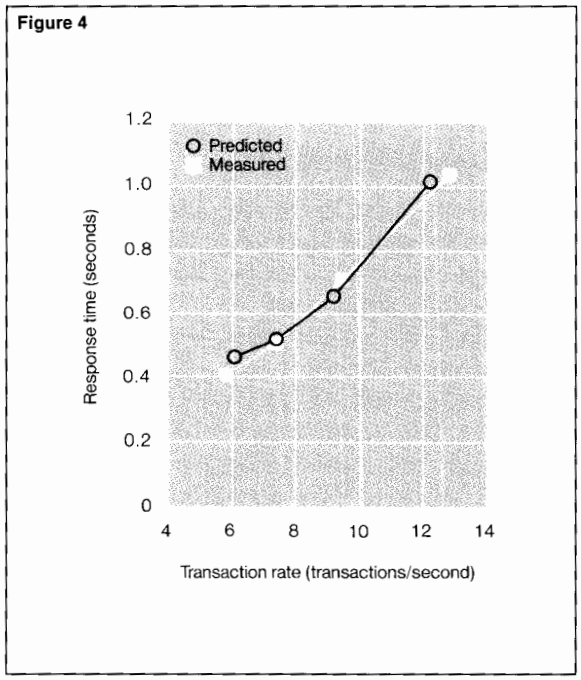


Table 1.
The data-base files used in the application benchmark.

Name	File type	Number of records	Record size (in bytes)
Account	Key-sequenced	1,200,000	100
Teller	Key-sequenced	1,200	100
Branch	Key-sequenced	120	100
History	Entry-sequenced	1 per transaction	50

Conclusion

The response-time prediction method just described is based on a simple simulation model for applications that use the PATHWAY transaction processing system. The model is small enough to be coded easily in a high-level language and has been observed to produce reasonably accurate results.

Although the model itself is quite simple, obtaining the parameters to feed the model requires an extensive amount of experimentation and knowledge about Tandem NonStop systems. Readers interested in implementing the method outlined here should note that response-time prediction is a difficult problem and that no easy shortcuts exist. This method reduces the task to manageable levels but does not make it trivial.

References

Chou, T.C.K., Oleinick, P., and Singh, A. 1984. *Language Directed Modeling*. 17th Hawaii International Conference on System Sciences, Honolulu, Hawaii.

Kobayashi, H. 1978. *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology*. Addison-Wesley.

Kosinski, S. 1984. The ENCORE Stress Test Generator for On-line Transaction Processing Applications. *Tandem Journal*. Vol. 2, No. 1. Tandem Computers Incorporated.

MacDougall, M.H. 1975. System Level Simulation. *Digital System Design Automation*, ed. M.H. Bruer. Computer Science Press.

Acknowledgments

The author would like to thank Peter Oleinick and Susan Whitford for their technical review and Rajiv Dhingra for his support.

Anil Khatri is a member of the Performance Group in Software Development. Before joining Tandem in 1983, he obtained a B.S. in Electrical Engineering from the Indian Institute of Technology at Kanpur and an M.S. in Computer Science from the University of Maryland.

Sizing Cache for Applications That Use B-series DP1 and TMF

The B-series release of Disk Process 1 (DP1) supports buffered writes for audited files. Because of this enhancement, users who protect their on-line transaction processing applications with the Transaction Monitoring Facility (TMF) should reevaluate their disk-cache configurations when migrating from A-series DP1 to B-series DP1.[1]

As most on-line transaction processing (OLTP) applications frequently perform random I/Os, users of TMF who convert to B-series DP1 should configure more disk cache to obtain a level of performance equal to (or perhaps better than) that they obtained previously. They should increase the size of the disk cache until the swap rate is adversely affected (subject, of course, to the amount of physical memory available).

Having configured a sufficient amount of cache, users of TMF will benefit from the many new features of the B-series software, such as autorollback, which dramatically improves the performance of crash recovery. Also, as a result of analyzing their application's use of cache and configuring an appropriate cache size, many users will reduce the disk activity required by their application, thus improving its performance.

Note that the disk-cache requirements of those users who do *not* employ TMF will remain the same when they convert to B-series DP1. They should obtain the same level of performance they obtained with the A-series software.

This article describes the performance analysis and subsequent changes in cache size required for one OLTP application protected by TMF when it was converted from A-series DP1 to B-series DP1. The article is intended as an example of (1) how to analyze application performance as related to DP1 cache size and (2) how to determine the appropriate cache size for applications using TMF that are to be converted to B-series DP1.

Methods of Analysis and Testing

First, the performance-analysis team ascertained that no changes were made to the application system other than converting it from A-series to B-series system software.

Then they used Tandem's XRAY performance analysis tool to measure the system's performance. The XRAY results revealed that the system was well balanced under both the A30 and B10 software. The CPU BUSY rate averaged between 45% to 50% under the A30 release and between 55% to 60% under the B10 release.

[1]Tandem software that is released to customers is identified by a release name. *A-series release* refers to Tandem software that is identified with an "A," such as the A20 software release or the A30 software release. *B-series release* refers to Tandem software that is identified with a "B," such as the B00 software release or the B30 software release.

Further analysis of the XRAY measurements showed that for the same work load, the disk I/O rates had increased significantly under the B10 software. The disk-cache size had remained unchanged at 48 pages per disk volume. CPU swap rates had not changed much, but cache-read hit rates were reduced. These observations caused the performance-analysis team to look more closely at the DP1 disk-cache management strategy and, specifically, at the difference between the strategies employed by the A- and B-series software.

As the application was a large one, porting the application to a stand-alone system for testing was ruled out. Instead, a small batch benchmark test that simulates only a part of the application transaction was run first. Only if its performance on A30 and B10 software was the same would testing of a larger version of the transaction be warranted. The benchmark was also designed so that the elapsed time to execute it took no longer than 10 to 15 minutes. This made it possible to run the benchmark under a variety of configurations when a longer benchmark would have made this more difficult.

Understanding B-series DP1 with TMF

The following is a brief explanation of the enhancements made to the disk process (in relation to TMF) in the B-series software releases. (For a more detailed explanation, see the following articles in the *Tandem Systems Review*: "TMF Autorollback: A New Recovery Feature," February 1985; and "Improvements in TMF," June 1985.)

A major enhancement to DP1 introduced in the B00 software release is the ability to buffer writes to disk for audited files. When writes are buffered, requests to modify audited files are replied to as having succeeded before the data is actually written to disk. Besides the obvious advantage of possibly causing fewer I/Os to disk, buffered writes also enable TMF to recover from crashes more efficiently by rolling the data base backward from the point of the crash (autorollback), rather than forward from the on-line dump (rollforward).

In order to be able to roll back the data base, the disk process is required to follow a write-ahead-log protocol; i.e., it must guarantee that the audit (log) records describing changes to the data base are written to disk before the updates are made to the application files.

The following are some terms useful in understanding the B-series software enhancements:

- *Dirty blocks* are cache blocks that have not been written to disk.
- *Clean blocks* are cache blocks that have been written to disk.
- *Replacement block* is the cache block that the disk process has chosen as the least-recently used (LRU) block.
- *Cleaning* means writing a dirty block to disk, thus making it clean.
- *Flushing* means writing audit records to disk.

Generally, for audited files, the disk process does not force changes to disk, but maintains images of the sections of the data base that have been modified in cache as dirty cache blocks. The one-page audit buffer in the disk-process cache holds the "before" and "after" images of the records being modified.

The sequence of events that occur when a record is modified for an audited file is briefly explained below. (Note that this is not intended as a complete explanation of how TMF works nor does it consider all possible exceptions.)

The audit buffer in the disk process has an audit-block number assigned for its location in the TMF audit-log file. The buffer occupies 2 Kbytes. The unmodified copy of the record is copied into this buffer. (Note that all audited files on the disk process would copy records to the same buffer.)

When the record is modified, the disk process does not force a physical write to disk, but maintains the image of the block that has been modified in cache as dirty cache blocks. The modified record is also copied into the audit buffer. The dirty cache block keeps a tag of the audit-block number into which it has copied the audited records. Changes made to dirty blocks in cache (write hits) save physical I/Os to disk and result in improved performance.

The dirty cache block cannot be written to disk as long as the "oldest" unflushed audit-block number is less than or equal to the tag number on the dirty block. This assures that audited information from the dirty block has been flushed to the TMF log file. This is defined as the write-ahead-log protocol.

Note that an audit-block flush can occur when the audit buffer is partially full and an ENDTRANSACTION occurs. When the partially full block is flushed, the oldest unflushed audit-block number is reset to null (indicating that all data in the audit buffer has been flushed), and the dirty blocks can be written out. However, the next audited write going into that partially full audit block again sets the oldest unflushed number to the current audit-block number. Hence, most of the dirty blocks having audited data in an audit block are flushed only when that audit block fills up and the disk process starts operating on the next audit block.

The dirty block is eventually written to disk when the write-ahead-log restriction is satisfied and one of the following occurs:

- The last opener of the file closes it.
- Control points are written.
- The memory page used for caching one or more of these blocks is requested by the memory manager.
- It is cleaned while the disk process is idle.
- A refresh is performed.
- The block is selected as the replacement block by the cache-replacement algorithm.

If a dirty block is selected by the cache-replacement algorithm as the replacement block, it may not be possible to free the block by writing it out to disk, owing to the write-ahead-log restriction. In this event, it is moved to a queue (called the *wait flush queue* in this article), and the disk process waits for the audit flush before cleaning it.

Therefore, while the queued dirty blocks are awaiting completion of the audit flush, the size of cache is effectively reduced by the number of blocks on the wait flush queue. Furthermore, since the dirty replacement blocks cannot be discarded to satisfy the current request for space in cache, the replacement algorithm may end up discarding some useful blocks (e.g., first- or second-level index blocks) from cache, thereby further degrading the performance of the application.

Comparing A- and B-series DP1

In the A-series software, modifications to audited files are always *write-through*; i.e., the requests to modify audited files are replied to as having succeeded only after the modified data is written to disk. Because of this approach, there are never any queued dirty blocks awaiting an audit flush, and hence, the effective cache size never decreases.

Rollforward may take hours to complete, while autorollback with B-series DP1 can be completed in minutes.

This approach has the drawback of inefficient crash recovery, however. A system may crash after the audited file has been modified on disk but before the audited data has been written to the TMF data audit disk. Thus, the only way TMF can recover from crashes is by rolling the data base forward from the last on-line dump (rollforward). Rollforward may take hours to complete, while autorollback with B-series DP1 can be completed in minutes.

Figure 1

```
$DATA04.TEST.PRIFILE
  TYPE K
  EXT ( 2 PAGES, 2 PAGES )
  REC 2018
  BLOCK  4096
  IBLOCK  4096
  KEYLEN  14
  KEYOFF  0
  ALTKEY ( "K2", FILE 0, KEYOFF 14, KEYLEN 6 )
  ALTKEY ( "K3", FILE 1, KEYOFF 20, KEYLEN 6 )
  ALTKEY ( "K4", FILE 2, KEYOFF 26, KEYLEN 6 )
  ALTKEY ( "K7", FILE 3, KEYOFF 600, KEYLEN 12 )
  ALTKEY ( "K8", FILE 4, KEYOFF 612, KEYLEN 26 )
  ALTFILE ( 0, $DATA03.TEST.ALT0 )
  ALTFILE ( 1, $SYSTEM.TEST.ALT1 )
  ALTFILE ( 2, $DATA01.TEST.ALT2 )
  ALTFILE ( 3, $DATA02.TEST.ALT3 )
  ALTFILE ( 4, $DATA04.TEST.ALT4 )
  PART ( 1, $DATA02, 4000 PAGES, 1000 PAGES)
  PART ( 2, $DATA01, 4000 PAGES, 1000 PAGES)
  PART ( 3, $SYSTEM, 4000 PAGES, 1000 PAGES)
  PART ( 4, $DATA03, 4000 PAGES, 1000 PAGES)

AUDIT
OWNER 1
SECURITY (RWEP): NNNN
MODIF: 11/19/85 8:56
EOF 0 (0.0% USED)
EXTENTS ALLOCATED: 0
```

LEVEL	TOTAL BLOCKS	TOTAL RECS	AVG # RECS	AVG SLACK	AVG % SLACK	PART
FREE	0					$DATA04
1	1	228	228.0	236	6	$DATA02
DATA	220	500	2.3	821	20	
FREE	0					
1	1	220	220.0	178	4	$DATA01
DATA	220	500	2.3	793	19	
FREE	0					
1	1	217	217.0	241	6	$SYSTEM
DATA	222	500	2.3	811	20	
FREE	0					
1	1	223	223.0	95	2	$DATA03
DATA	223	500	2.3	803	20	
FREE	0					

Figure 1.
The File Utility Program (FUP) information for the primary partition used in the benchmark.

The Benchmark

In analyzing the customer application, the performance-analysis team noted that the major part of the transaction involved updates to records in a key-sequenced file. These records had between three to five alternate-key fields, which were also modified during the update.

In the benchmark, a primary key-sequenced file contained records having five alternate-key fields. An application process sequentially read and updated all the records in this primary file. During these updates all five alternate-key fields were modified. This caused the file system to read, delete, and write the alternate-key records in the five respective alternate-key files. The block and record sizes used were the same as those used by the application.

The benchmark was run on a four-processor NonStop TXP system containing eight mirrored disk drives. The primary key-sequenced file had four partitions. Each disk on which a partition resided had its disk "primaried" to one processor. The five alternate-key files were placed on five different disks. Three of the processors each had one disk primaried to it. The fourth processor had two of these disks primaried to it.

Four application processes were run, one on each processor. Each of these application processes operated on only one primary partition, but the record updates caused activity on all five alternate-key files.

The tests were run under versions A30 and B10 of the GUARDIAN operating system, with DP1.

The File Utility Program (FUP) information for the primary partition is given in Figure 1. Note that the BLOCK and IBLOCK sizes on all the alternate-key files were 4096 bytes.

Figure 2 illustrates the hardware configuration and the distribution of the files on the system. The four application processes were named $FB1, $FB2, $FB3, and $FB4. PART1 through PART4 were the four partitions of the key-sequenced file. ALT1 through ALT5 were the five alternate-key files.

Note that process $FB1 sequentially read and updated 500 records in the partition (PART1) existing on $DATA02. These updates caused activity on all five alternate-key files. Similarly, process $FB2 sequentially read and updated 500 records in the partition (PART2) existing on $DATA01. The same occurred for processes $FB3 and $FB4. All application processes were started concurrently. For consistency, the data base was reset after every test run.

Results

While the performance-analysis team would have liked to measure the average response time required to update a record in the primary file and all the corresponding records in the alternate-key files, obtaining this information would have required substantial effort in

setting up the benchmark. As a more practical alternative, the team chose to measure the *sum of the elapsed time for each application process to complete 500 updates.*

The XRAY reports studied closely by the team were the Disc Device and Disc Open reports. Physical I/O activity was the most important measure, as it represents the effectiveness of the disk cache. Note that the team was not interested in the CPU BUSY times, as their earlier atomic testing of the A- and B-series software had not shown any significant variation.

For the initial testing, the minimum disk cache was configured for each of the disk volumes. (This is 13 pages of disk cache per disk volume for the A30 release and 15 pages per disk volume for the B10 release.) The smallest cache sizes were chosen since the team was interested in studying the behavior of disk cache under cache pressure.

In B-series DP1, two pages of cache are reserved for unaudited files. Hence, in the above configurations, both the A30 and B10 versions had 13 pages of cache available for audited files.

Figure 2

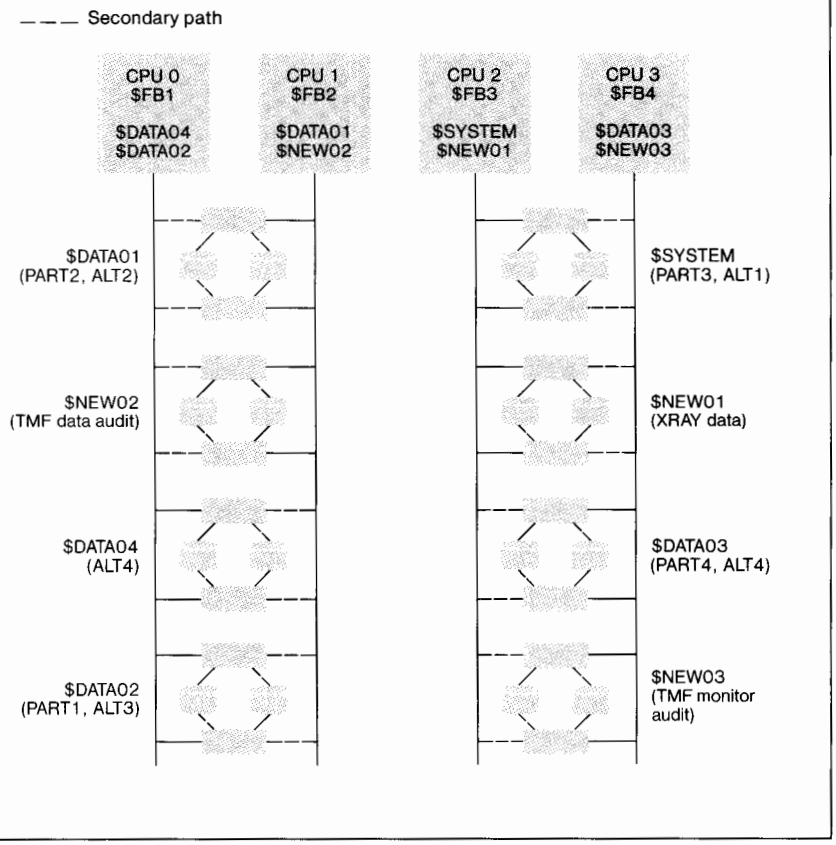


Figure 2.
The hardware configuration and distribution of files for the benchmark. The four application processes are $FB1, $FB2, $FB3, and $FB4. PART1 through PART2 are the four partitions of the key-sequenced file. ALT1 through ALT5 are the five alternate-key files.

Performance-measurement Counters

The team used the following XRAY performance-measurement counters:

- *Physical writes* represents the total write operations to a disk, as listed on the XRAY Disc report.
- *Cache-write hits* represents the total cache-write hits, as listed on the XRAY Disc Open report. A cache-write hit occurs when a write request is applied to a dirty cache buffer, thus avoiding an additional physical write to the disk.
- *Physical reads* represents the total read operations to a disk, as listed on the Disc report.
- *Cache-read hits* represents the total cache-read hits, as listed on the Disc report. A cache-read hit occurs when a read request is satisfied by a cache read, thus avoiding a physical read to the disk.
- *CBKS0 dirty* represents the average number of dirty cache pages. (Note that the team chose to measure this in pages of 2048 bytes instead of in sectors of 512 bytes as the Disc report measures it. Hence, they divided the value of the sector counter on the Disc report by 4 to represent the counter in pages.) A cache block is marked dirty when a buffered write operation stores data in a block.
- *MAX0 dirty* represents the highest number of dirty pages occurring within the measurement window, as listed on the Disc report. The value of this counter is affected by the availability of physical memory and by the number and pattern of buffered writes.

The following counters were devised especially for this investigation and are not available on XRAY reports:

- *Wait flush queue* represents the average number of dirty cache pages that have been selected for replacement but cannot be written to disk because of the write-ahead-log restriction.
- *Maximum wait flush queue* represents the highest number of dirty cache pages in the wait flush queue occurring within the measurement window.

For simplicity, the benchmark results reported in this article are for the two busiest disks only.

Table 1.
A comparison of application performance under A30 and B10 DP1 when TMF is not used (Test 1).

	A30 DP1		B10 DP1		Percentage change from A30 to B10
	$DATA01	$DATA03	$DATA01	$DATA03	
Physical writes	4560	4560	4572	4570	+ 0.2%
Physical reads	3809	3397	2936	3734	-7.5%
Cache-read hits	6989	7471	8035	7244	+ 5.7%
Total elapsed time (secs)	2066		1911		-7.5%

Table 2.
A comparison of application performance under A30 and B10 DP1 when TMF is used (Test 2).

	A30 DP1 and TMF		B10 DP1 and TMF		Percentage change from A30 to B10
	$DATA01	$DATA03	$DATA01	$DATA03	
Physical writes	4475	4473	3830	3860	-14.1%
Cache-write hits	0	0	747	713	
Physical reads	3742	4007	6164	6086	+ 58.1%
Cache-read hits	6913	6655	4231	4240	-37.6%
CBKS0 dirty (pages)	0	0	5.45	5.87	
MAX0 dirty (pages)	0	0	12	12	
Wait flush queue (pages)	0	0	3.2	3.2	
Maximum wait flush queue (pages)	0	0	12	12	
Total elapsed time (secs)	2551		3413		+ 33.8%

Test 1

To help isolate the problem, the performance-analysis team ran the first test with TMF off. Cache was configured as 13 pages per volume for the A30 software and 15 pages per volume for the B10 software. The results of Test 1 are listed in Table 1.

The team observed that no significant difference in performance could be seen for A- and B-series software. The 7.5% decrease in elapsed time, 7.5% decrease in physical reads, and 5.7% increase in cache hits can be attributed to the fact that, under B10, two extra pages of disk cache per volume were configured.

This test verified that no major changes in performance occur between A- and B-series DP1 for unaudited files.

Test 2

The second test used the same configuration as that for Test 1 except that TMF was on and audited files were used. The results are listed in Table 2.

In this test the performance-analysis team observed that elapsed time increased by more than 30% when the B10 software was run, confirming that performance was adversely affected when B-series software was used with DP1 and TMF audited files.

Using the results in Table 2, the team found that the number of physical reads increased by 58.1% and the number of cache-read hits decreased by 26.8%.

Of the 15 pages of disk cache per volume configured for the B10 software, two pages are reserved by DP1 for unaudited file operations. This makes 13 cache pages available for audited operations, the same number available with the A30 software.

Of the 13 pages, one page is used by the audit buffer to collect the audited data for its disk process. This leaves 12 cache pages for audited file operations. The team observed that, on both disks, the maximum wait flush queue was 12. This was definitely a problem. It meant that during the test the size of the cache was effectively reduced to 0 pages at least once and possibly more than once.

Volume $DATA01 contained the files PART2 and ALT2, which were key-sequenced files. The maximum wait flush queue on $DATA01 reached 12, meaning the index blocks of these files were also picked for replacement from cache during the test. The same was true on volume $DATA03.

The above problem occurred because each update to an alternate-key file dirtied two data blocks, a delete and an insert. These dirty

blocks were not available for cleaning until the disk process operated on the audit block number that was greater than the tag number on the dirty blocks.

Note that the record length of these alternate-key records was only 22 bytes; hence, the delete and insert of a record in this file generated only about 100 bytes of audit information for the 2-Kbyte audit buffer. Thus, three application processes modifying this file would dirty six blocks (the entire cache of 12 pages) and would have generated only 300 bytes of audit data. This would reduce the available cache to 0 pages.

The performance-analysis team concluded from Test 2 that application performance was degraded when the B10 software was run because dirty blocks remained in cache until the audit-block number in cache was greater than the tag number on the dirty blocks.

Test 3

In the third test, the team wished to see if increasing the disk cache per volume would improve the performance of the benchmark under the B-series software. The configuration used in this test was the same as that in Test 2, except that the disk-cache configuration for the B-series software was increased to 25 pages per volume.

In Table 3, the results for the the B10 software are listed for Test 2 and Test 3. In both tests, TMF was on. Cache was configured as 15 pages per volume for the B10 software in Test 2 and as 25 pages per volume for the B10 software in Test 3 (although only 23 pages could be used for audited files).

The performance-analysis team observed that the total elapsed time of the benchmark decreased by 28.9% when the size of the disk cache for the B10 software was increased.

Also, the total elapsed time for the B10 software when 25 pages of cache was configured was 2425 seconds, while the total elapsed time for the A30 software when 13 pages of cache was configured was 2551 seconds. Thus, the B10 software with 25 pages of cache was 5% faster than the A30 software with 13 pages of cache.

The team concluded that to obtain the same performance *for this benchmark* under A- and B-series software when TMF was used, more disk cache had to be configured for the B-series software.

Table 3.
A comparison of application performance under B10 DP1 when TMF is used and cache is configured as 15 pages (Test 2) and 25 pages (Test 3).

	B10 DP1 and TMF, 15 pages of cache (Test 2)		B10 DP1 and TMF, 25 pages of cache (Test 3)		Percentage change from 15 to 25 pages
	$DATA01	$DATA03	$DATA01	$DATA03	
Physical writes	3830	3860	1705	1704	-55.6%
Cache-write hits	747	713	2865	2819	+289.0%
Physical reads	6164	6086	2059	2051	-66.5%
Cache-read hits	4231	4240	9005	9006	+112.0%
CBKS0 dirty (pages)	5.45	5.87	13.3	13.4	+135.0%
MAX0 dirty (pages)	12	12	20	20	+66.7%
Wait flush queue (pages)	3.2	3.2	1.84	1.57	-46.7%
Maximum wait flush queue (pages)	12	12	20	20	+66.7%
Total elapsed time (secs)	3413		2425		-28.9%

Conclusions

The performance-analysis team formed the following conclusions from this investigation.

Applications that do not use TMF will not experience performance problems when they are converted to the B-series software, as their disk-cache requirements will not change. Applications that use TMF, however, will require a reevaluation of their disk-cache configuration when they are converted from A- to B-series DP1.

To compensate for the effective reduction of cache size caused by dirty blocks, systems running B-series DP1 with TMF may require a larger cache (as specified with the CACHE-PAGES modifier in SYSGEN) than that used under A-series DP1.

How Much Is Enough Cache?

Users who configure "enough" cache will not experience performance problems by migrating to B-series software. Exactly how much cache is enough is dependent on the nature of the application. Some of the factors that determine the amount of cache required are locality of reference, number and type of files that are open on the disk volume, and the sizes of records, blocks, and files.

Unfortunately, no simple rule exists for determining cache size. The two most important factors, however, are the amount of physical memory on the CPUs containing the primary and the backup disk processes, and the amount of file activity on the disks.

Physical Memory

Physical memory is shared by the memory manager and the disk cache for each disk process (primary and backup) on the CPU. The ideal division of memory allots the memory manager all of the memory it needs (no extra) and allots the rest as disk cache.

If the memory manager has too much memory, disk processes may perform unnecessary disk I/Os, while if it has too little, it performs unnecessary swap operations. (Swaps are also disk I/Os and, therefore, expensive.) In general, a disk process's cache should be increased just to the point at which it begins to affect the swap rate in its primary or backup processor. In some instances, additional physical memory may also be needed in the processors.

A caveat: The sum of the cache sizes of all the disk processes (primary and backup) in a processor must not exceed 70% of the swappable memory available in that processor. If caches are too large, processor halts may occur as a result of insufficient physical memory.

Some concern has been expressed that increasing disk-cache sizes will result in less memory being available for user processes. It is true that increasing disk cache sizes will increase the swap rate in a heavily loaded system; however, the disk-cache sizes specified by the CACHEPAGES modifier in SYSGEN define the virtual address space used by the disk processes for their caches. It does not mean that a corresponding amount of physical memory is used.

Physical pages are allocated under virtual address space as needed. When the memory manager finds that the replacement page it has chosen according to its replacement algorithm is used by a disk process for holding cache blocks, it requests the disk process to release the page. Thus, in the primary CPU, the protocol between the memory manager and disk processes ensures that allocation of physical memory for disk caching is dynamically dependent on memory pressure.

File Activity

Several aspects of file activity can affect disk-cache size. The number of active files open on the disk is one. A second aspect comprises the following file characteristics:

- Locality of references.
- Frequency of references.
- File size.
- Size of index and data blocks.
- Record size.

The effect of both of these aspects is discussed below.

Number of Active Files Open on the Disk. Each of the active files on a disk process has its own cache requirement. The sum of the cache requirements for all active files is the preferred cache size for that disk, but because physical memory may be limited, it is not always possible to configure the preferred cache size. For systems in which the size of physical memory is a limiting factor, some disks will have to be configured with caches smaller than the preferred size. The compromise should be made in favor of disks that have files with higher reference frequency.

File Characteristics. When a key-sequenced file is accessed randomly, one or more index blocks in addition to the data block must be accessed for each I/O operation. When enough cache to hold the index blocks for the file is configured, disk I/Os associated with the index levels can be avoided, resulting in improved performance. The cache size required depends on the number and size of the index and data blocks.

When an entry-sequenced or relative file is accessed randomly, if enough cache to hold a certain percentage of the file (e.g., 30%) is configured, the chance of a cache hit is substantially increased, thus improving performance. In the event of a cache miss, however, an entry-sequenced or relative file requires one I/O, whereas a key-sequenced file may require more than one. Therefore, if the frequency of references is approximately the same, the cache requirements for a key-sequenced file should have precedence over the requirements for other types of file, when more than one volume is connected to a CPU.

When any file is accessed sequentially, the information is accessed only once, so the cache requirement for the file is minimal. The same cache blocks can be used repeatedly to bring data from disk as the previous information is discarded. Even for a key-sequenced file, minimal cache should keep the required index blocks in cache until they are replaced by the index blocks required for the next set of data blocks.

Updates to files with small record sizes, e.g., 100 bytes, generate only a small amount of audited data. It would require a large number of updates to fill up the 2-Kbyte audit buffer, and until then, all the dirty blocks would effectively reduce the cache by that number of blocks. To compensate for this effect, the cache size should be increased with B-series software.

For small files, if enough cache to hold the entire file is provided, a significant number of disk I/Os can be avoided, resulting in improved performance. Note that providing cache does not guarantee that the file remains in cache. Unless the small file is busy, its blocks are likely to be swapped out in favor of blocks of busier files.

General Guidelines for Sizing Cache

Although cache is configured for the primary disk process, the system automatically configures the same amount of cache for the backup disk process. The impact of disk-process cache size on both CPUs should be considered. It is possible for the backup CPU to run out of memory without memory pressure existing in the primary CPU.

The disk cache (primary and backup disk processes) for a CPU should not consume more than 70% of that CPU's swappable memory. Inadequate memory for the memory manager can cause the CPU to halt.

When the file activity on a disk is analyzed, only those files that are heavily used should be considered. Infrequently used files have little effect on performance and should not be considered when cache is sized.

Generally, for DP1, cache can be increased just to the point at which it starts affecting the swap rate in its primary or backup processor. Use the file activity information for each disk to determine which disk process requires an increase or decrease in cache.

In brief, there is no simple rule to determine "correct" cache sizes for DP1 with TMF. File activity should be used as the basis for initially configuring cache size. This size should then be modified, based on performance measurements such as those outlined in this article.

References

Lemberger, T. 1985. Improvements in TMF. *Tandem Systems Review*. Vol. 1, No. 2. Tandem Computers Incorporated.

Pong, M. 1985. TMF Autorollback: A New Recovery Feature. *Tandem Systems Review*. Vol. 1, No. 1. Tandem Computers Incorporated.

Acknowledgments

A number of people contributed substantially to understanding the application, setting up the benchmark, and analyzing the results. Among them were Sanjay Laud and Gary Tom of Software Development, Walt Gomsi and Anita Tucker of the Silicon Valley office, and Mike King of Northwest Software Support.

Praful Shah joined Tandem in June 1984. Since then he has worked with the Performance Group in Software Development on performance studies related to DP2, DP1, TMF, processors, and peripherals. Before joining Tandem, he worked in a performance group for another mainframe vendor. Praful has an M.S. in Computer Science from Pennsylvania State University and a B.S. in Electrical Engineering.

Plated Media Technology Used in the XL8 Storage Facility

Tandem's new XL8™ disk storage subsystem is composed of eight disk drives, each holding 525 Mbytes, and can store 4.2 gigabytes of data in only six square feet of space. The disk drive, combining low cost per Mbyte with outstanding performance, incorporates the latest in disk technology, such as Whitney[1] recording heads, high bandwidth read/write channel, high-density VLSI, 2-7 run-length limited data encoding, and plated media.

Figure 1

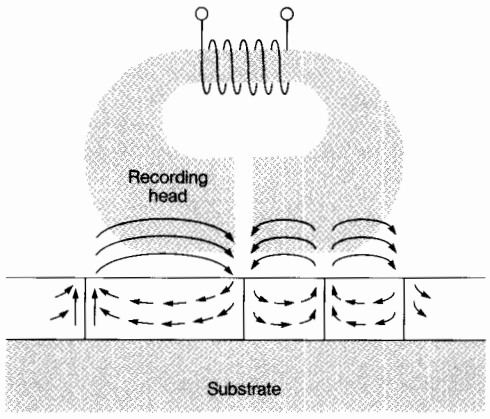


Figure 1.
Magnetic recording. The recording head is energized by the current flowing through the windings. The magnetic field produced orients the particles in the same direction.

[1]Whitney is the name commonly associated with the second-generation Winchester technology used in the IBM 3370 disk drives first announced in 1979.

This article is concerned with the plated media. A second article, immediately following, describes 2-7 run-length limited data encoding.

This article discusses:

- Basic concepts of magnetic recording.
- The benefits of increased storage density and product reliability.
- Oxide-coated media as compared to thin-film media, including the thin-film media manufacturing process, and electrical and magnetic characteristics.
- Head and plated media specifications.
- Recording density trends.

Basic Concepts of Magnetic Recording

The magnetic recording process used in mass data storage disk drives is based on the same fundamental principles as audiocassette tapes and VCR machines.

In the recording process, a film of magnetic material coated onto a carrier medium is passed under a recording head that is energized by electrical current through its winding. The magnetic flux created by the head orients the direction of magnetization on the medium, forming magnetic domains that represent bits of data.

In the reading process, the medium is again passed under the head. A voltage is induced in the head by the change in flux direction as it passes over each domain previously recorded. This voltage is sensed by amplifiers mounted very close to the heads, creating a read signal. The signal is then filtered, amplified, modulated, and encoded into synchronous digital data. (See Figure 1.)

In disk drives, the recording medium is usually coated onto round rigid aluminum platters. A stack of one or more platters is driven by a spindle motor. When the motor stops, the heads rest on a special landing zone on the disk. As the motor starts to rotate and drag the ambient air with it, the aerodynamic design of the head allows it to take off and fly, cushioned on the air passing underneath it. This flying height is measured in microinches. In fact, the head/media gap is so small in relation to the head size that it has been compared to flying a Boeing 747 several inches off the ground. When the disks stop, the heads land back on the disks. This process is known as contact start-stop mode. During the life of a disk drive, the head may land on the media many times. Increased media hardness and a carefully chosen lubricant are needed to avoid damaging both the head and the media.

More Storage Capacity in Less Space

Areal recording density has increased 3000-fold in the past three decades.

The earliest hard disks were 27 inches in diameter with 100 bits per inch (bpi) and 20 tracks per inch (tpi). This yielded a recording density of 2000 bits per square inch (bpsi).

Twenty years ago, the IBM 1301 was capable of recording at 26,000 bpsi. Then, in 1974, the Winchester heads flying at about 20 microinches, advanced linear density to 10,000 bpi (6 million bpsi).

The state of the art was again advanced in 1982 when IBM introduced the 3380. The 3380 uses thin-film head and oxide media to achieve a bit density of 15,000 bpi and a track density of 800 tpi to store 12 million bpsi. The head flying height is about 10 microinches.

Tandem's XL8 has a linear density of 18,600 bpi and a radial density of 1000 tpi. This means 18.6 million bits can be stored in one square inch of space; i.e., over 1300 pages of double-spaced type-written document can be stored in an area the size of a postage stamp.

Figures 2 and 3 show the development in recording density. Table 1 compares the IBM 3350 and 3380 technology to Tandem's XL8 disk storage subsystem.

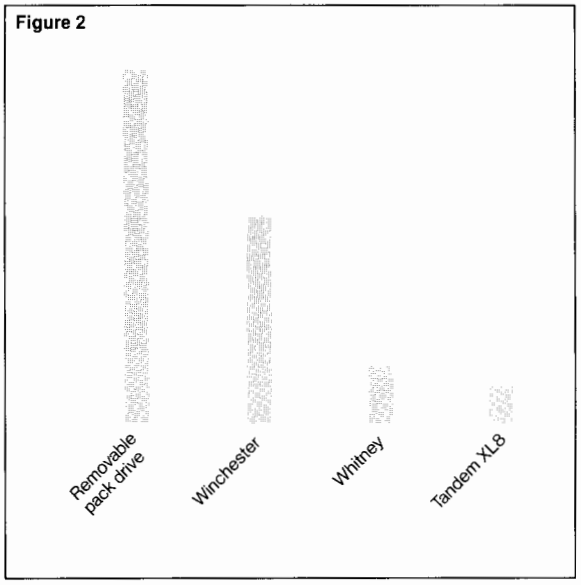


Figure 2.
Disk media required to store 10 million bits of data. The bars represent the area required to store the same amount of information using different recording technology developed over the last ten years.

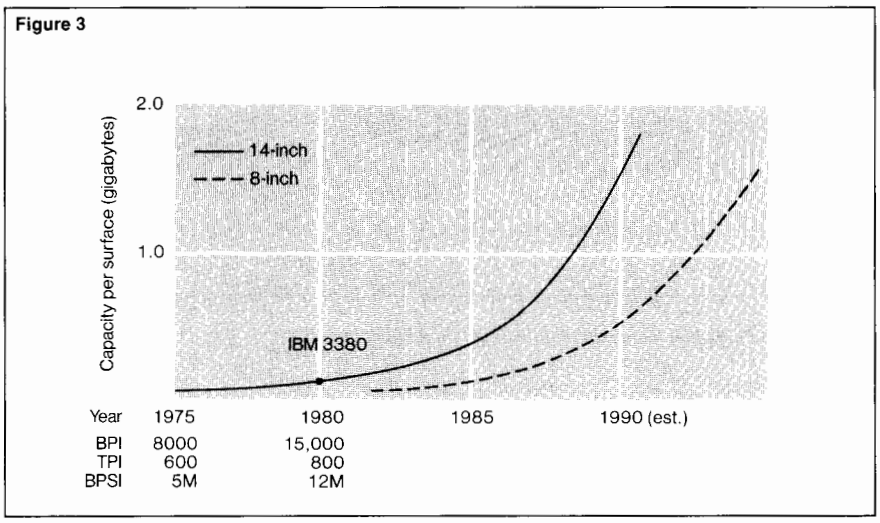


Figure 3.
Recording capacity per disk surface. This graph shows the exponential growth in storage capacity per disk in the last decade. It also compares the capacity of disk platters of two popular diameters: 8 inches and 14 inches.

Table 1.
A comparison of IBM and Tandem disk technology.

	1974	1981	1985
	IBM 3350 (oxide)	IBM 3380 (oxide)	Tandem XL8 (plated)
BPI	6425	15,000	18,600
TPI	480	800	1000
Areal density (million bpsi)	3	12	18.6
Mbytes/spindle	317	1260	520
Head flying height (μ inches)	18	10	10
Average access time (msec)	25	16	15
Data transfer rate (Mbytes/sec)	1.2	3	1.8

Improved Areal Recording Density

Areal recording density is improved by increasing radial density (tpi) and linear density (bpi). Both head and media innovation play a vital part in its advancement. Head development concentrates on reducing the width of the gap between the magnetic poles and the gap between the head and the media. As these gaps are narrowed, smaller magnetic domains can be formed, thus increasing linear density.

Figure 4

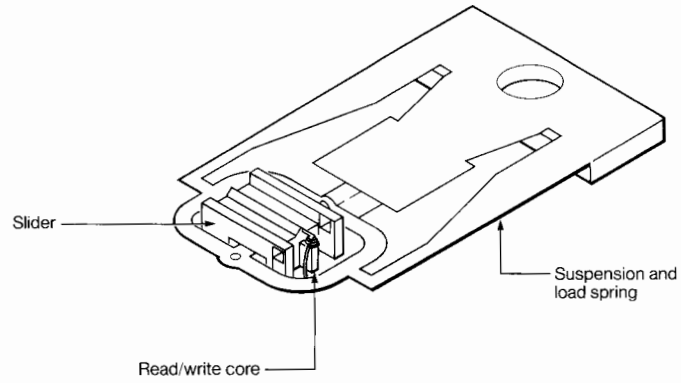


Figure 4.

Major components of a head assembly. The suspension serves as a mounting base for the head element and provides a preloaded force to it. The slider acts like the wings of an airplane to provide aerodynamic lift to the head element. The read/write core constitutes the rear end of the head element and the windings on it are shown.

There are two complementary requirements in new media development:

- The thickness of the magnetic film must be reduced. A thinner layer works with a lower flying head to allow for saturation[2] of the recording surface as it is magnetized.
- Better magnetic properties such as increased coercivity[3] and remanence[4] are required for signal quality.

Heads

Heads consist of a coil and magnetic poles and are usually made of a ferrite read/write element mounted on a slider. The slider is either made from ceramic (for composite head) or bulk ferrite (for monolithic heads). The slider is shaped by precision grinding and lapping, with glass bonding forming the read/write gap. Figures 4 and 5 show the construction of the ferrite heads. The drawback is in frequency response. Improved manganese-zinc ferrite heads have a maximum frequency response of about 15 MHz.

The XL8 disk storage subsystem uses ferrite heads. Because of the advanced data encoding method, the heads are writing and reading flux changes at a rate of 10 MHz. (See the accompanying article, "Data Encoding Technology Used in the XL8 Storage Facility.")

Media

Media can be broadly classified into two categories: oxide media and thin-film media.

Oxide Media. Oxide, particulate, thick-film, and coated media are all terms for the type of media used for the past 20 years. The coating material is made by mixing iron-oxide particles with an organic binder and applying it over the aluminum substrate. This forms a coating 20 to 50 microinches thick. The disk is then polished to a high luster and overlaid with a carefully controlled film of special lubricant. Typical coercivity for oxide media

[2]The magnetic layer is saturated when the magnetic poles of all the particles are lined up in the same direction. An unsaturated layer produces a weaker read signal and has a shorter shelf life as the magnetic energy stored is being used to induce the remaining magnetic particles to line up in the same direction.

[3]Coercivity, represented by the symbol Hc, is the magnetic field strength of opposite direction required for reduction of the remanence to 0. It is measured in units of Oersted (Oe).

[4]Remanence, represented by the symbol Br, is the magnetic induction that remains in a material after the removal of the magnetizing force. It is measured in units of Gauss (Ga).

is 300 to 500 Oe. Recent attempts to extend its recording density involve doping the traditional iron-oxide particles with cobalt or using alternate particles. This results in a deep brown color and is called "chocolate" media.

Thin-film Media. The thin-film media's magnetic film is usually only 3 to 10 microinches thick. It can be applied by either electrochemical plating or sputtering (molecular deposition in a partial vacuum). Since the total electric charge in a plating process or the deposition time in the sputtering process can be precisely controlled, it is possible to produce recording films of any thickness, even down to 1 microinch. Coercivity characteristics are also higher, in excess of 600 Oe.

Plated Media Manufacturing Process

There are four basic steps involved in making a plated disk:

1. A blank aluminum disk is processed into a substrate suitable for plating.
2. A layer of electroless nickel is plated onto it.
3. A thin film of magnetic material is deposited on top of the nickel layer.
4. A carbon overcoat is applied as lubricant.

The disk manufacturing process must be carefully controlled from the very beginning. The blank substrate is first checked for uniform thickness and flatness. A slight variation in these dimensions will cause problems in head flight and even crashes. The inner and outer diameters are also checked for tolerances. Then the blanks are subjected to heat treatment for an extended period of time. This annealing process relieves metal stress. The blanks are then ground down through several passes to even smoothness.

Figure 5

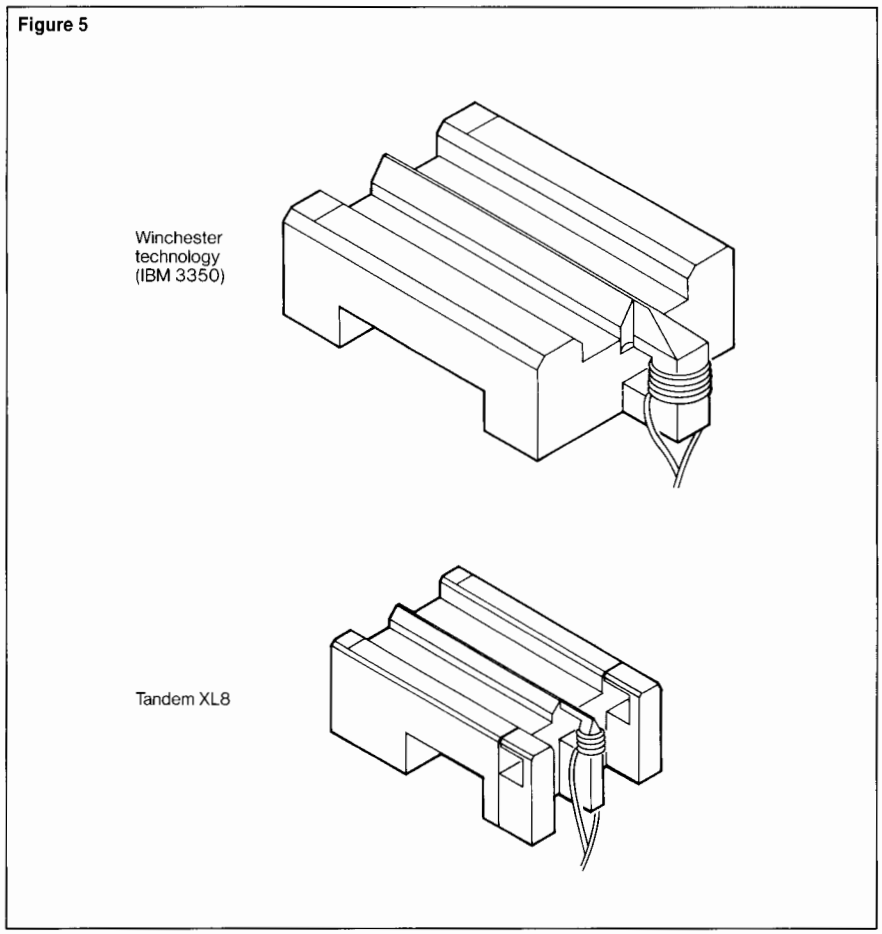
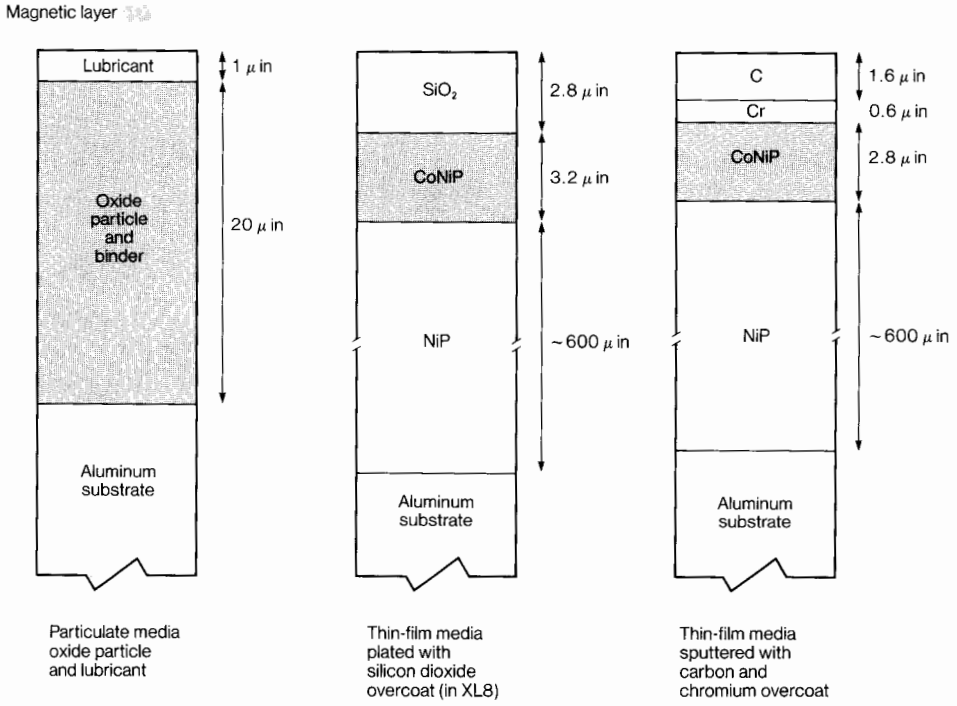


Figure 5.
Two generations of Winchester recording heads. The two types of heads are drawn to the same scale to show the relative sizes.

Figure 6.

Comparison of particulate versus thin-film media thickness. The magnetic layer of the particulate (20 microinches) is about seven times as thick as that of the two thin-film media (3 microinches).

Figure 6



The second manufacturing stage, applying the nonmagnetic nickel phosphorus layer, serves three functions. This layer provides the hardness needed for contact start-stop operation, seals the substrate to avoid corrosion, and provides better adhesion for the magnetic layer. It is typically several hundred microinches thick and is plated on.

After the nickel layer is polished, it is ready for the plating of the magnetic film. This film is usually less than 5 microinches thick and is made of cobalt or nickel or both, plus a certain amount of phosphorus.

Table 2.
Specifications of the thin-film disk and heads used in the XL8 disk storage subsystem (per NEC Information Systems).

Thin-film disk	Specifications
Mechanical data	
Outer diameter	230 mm, 9.05 inch
Inner diameter	100 mm, 3.94 inch
Durability (contact start/stop cycles)	20,000 times/min
Magnetic characteristics	
Coercivity	700 Oe
Remanence	7000 Gauss
Media thickness (Cobalt Nickel Phosphorous)	0.08 μM, 3 μ inch
SiO_2 overcoat thickness*	0.08 μM, 3 μ inch
Electrical data	
Resolution	79.5% min
Overwrite residue	-29 dB max
Signal to noise	32.8 dB min
Head	
Mechanical data	
Gap length	0.8 μM
Flying height (inner track)	0.28 μM, 11 μ inch
Electrical data	
Inductance	4.8 μH
Readback amplitude	24 mv p-p min

*NEC uses a polar fluorocarbon lubricant that chemically adheres to the SiO_2 layer to enhance smooth contact start-stop motion and mechanical durability.

Finally, the lubricant is applied and the disks are checked for surface defects. The choice of lubricant is extremely important as it determines the performance and life of the disk. The formula and process used are often proprietary to the manufacturer. Most manufacturers use a carbon overcoat, while some have developed a proprietary silicon dioxide lubricant. A chromium layer is sometimes applied before the lubricant is applied to increase corrosion resistance. The result is a hard, corrosive-resistant disk having few defects, a disk well suited for high-density recording.

Figure 6 shows the relative thickness of the different layers. Figure 7 illustrates the process that produces the plated medium used in the XL8 disk drives. The lubricant used is silicon dioxide.

Electrical and Mechanical Characteristics

The thin-film media are superior in several ways to their coated counterparts. First, the decreased thickness of the magnetic layer contributes to a higher recording density. Since the recording head's magnetic flux does not have to penetrate as deeply in order to saturate the layer, smaller magnetic domains can be formed with a thinner layer.

Table 2 shows some specifications of the thin-film disk and heads used in the XL8 disk storage subsystem.

Second, the material used during the manufacturing process enhances the magnetic and mechanical properties. The cobalt and nickel in the recording film increases the remanence value to about eight times that of the oxide media. This gives better signal definition. The phosphorus raises the coercivity to 70% greater than oxide media. This increases the signal-to-noise ratio. The nickel undercoat adds hardness to the disk. This increases durability and decreases handling and shipping damage.

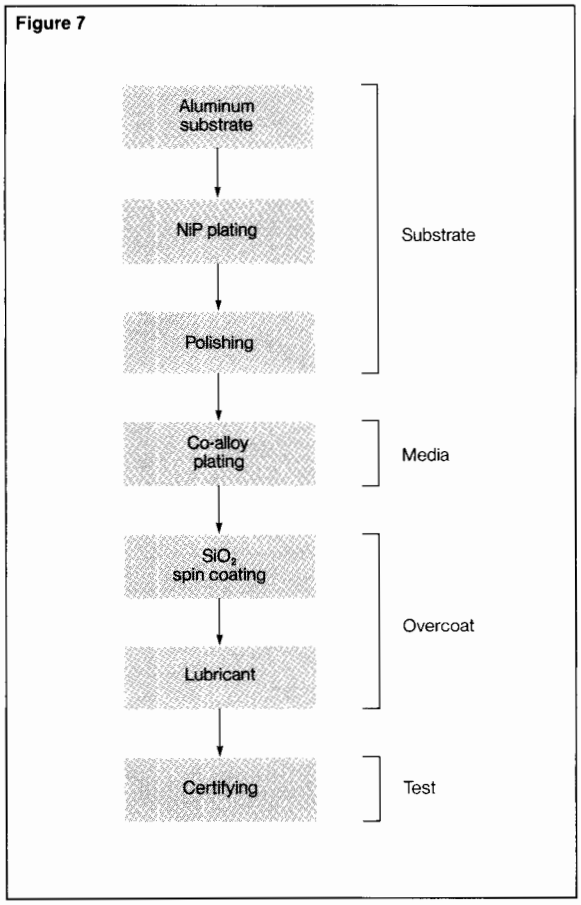


Figure 7.
Plated disk production process. This flow diagram shows the major steps of plated disk production.

Tests performed by disk manufacturers show almost no signal degradation after 100,000 contact start-stops. Assuming two start-stop cycles per day, this represents a media life of 12 years.

Figure 8

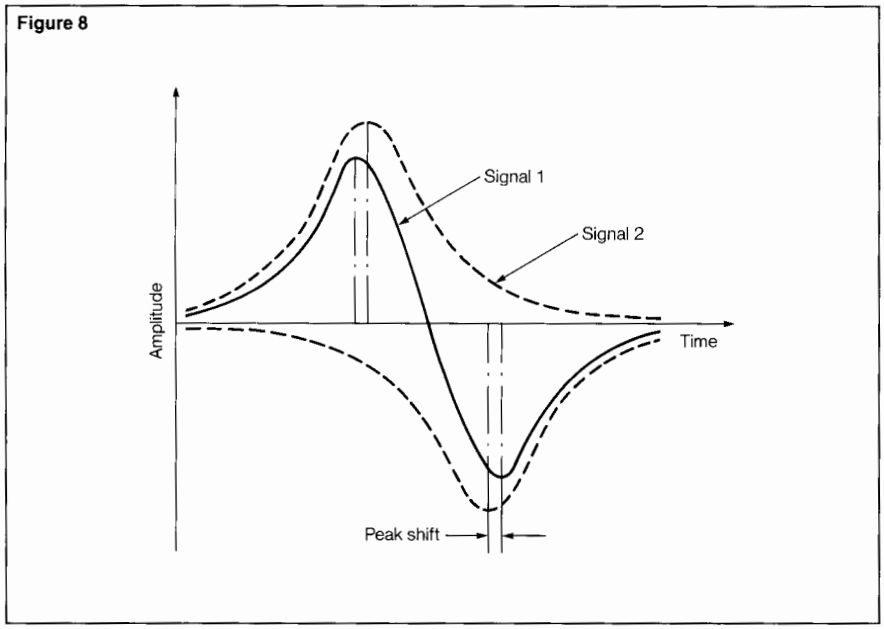


Figure 8.

Peak shift phenomenon. When two magnetic flux transitions are recorded close to each other, they have a tendency to repel each other. The resultant waveform read-back exhibits a phenomenon known as peak shift. Here the two peaks shift away from their respective nominal positions and their amplitudes decrease as well.

Another important media characteristic that directly affects data error rates is the phenomenon known as *peak shift*. Peak shift is caused by the superposition of signals from adjacent flux transitions. As bit density increases, so does the effect of superposition.

Figure 8[5] shows how the peaks of the two adjacent bit cells shift away from each other as their amplitudes decrease in much the same way as like poles of a magnet repel each other. In order to read a flux transition in its true position, it must be detected within an interval known as the bit-cell time. When the peak shift gets too large, the individual peak becomes hard to identify and consequently the data error rate increases rapidly.

Figure 9 shows the relationship between peak shift value as a percentage of the bit-cell time and recording density expressed in terms of flux transitions per inch. It compares the oxide media to thin-film media using ferrite (MnZn) heads. As the curve on the left indicates, peak shift for a ferrite head and oxide media increases sharply with recording density. Peak shift characteristics of thin-film disks show greater capability for high-density recording. This graph demonstrates the potential for going beyond 15,000 flux transitions per inch.

Table 2 contains some specifications of the thin-film disks and heads used in the XL8 disk storage subsystem. The electrolytic plating process provides superior defect performance and excellent signal resolution at high density. As more media manufacturers gain experience and the ability to control the process, yield, availability, pricing, and consistency of quality will improve. In the future, more disk drive manufacturers will make use of this technology which is currently available in Tandem products.

[5]Figures 8 and 9 originally appeared (in a slightly different form) in an article entitled "Thin-film Disks Drive Densities to New Highs," by Jack Taranto in *Electronics*, April 21, 1982.

Plated Media Potential

While 18 million bpsi is the maximum for oxide media, it is the floor for plated media. It is expected that within three years, linear density will be increased to 25,000 bpi as radial density is advanced to 1600 tpi, yielding 40 million bpsi.

The other thin-film process used to increase density is called sputtered media. The sputtered magnetic film can be adapted more easily to vertical recording. Since the sputtering process is similar to semiconductor wafer processing, most announced sputtered media are 5¼ inches in diameter or smaller. No vendor has yet announced a larger sputtered-media disk. Developing a larger disk would require very expensive tooling changes and the process is much more difficult to control.

On the other hand, a few vendors have mastered the electrochemical plating process with consistent quality and high yield. The overcoating has proven to be durable and long-lived, even under severe environmental conditions. Defect control is also better. Larger diameter disks can be mass-produced with the plated process, thus bringing down substantially the cost per Mbyte of storage.

Conclusion

The XL8 disk storage subsystem is a significant addition to Tandem's fault-tolerant computing equipment. Its plated media characteristics have been measured and verified in our disk device development labs. These attributes not only make the plated media an outstanding choice for high-capacity, high-density storage, they also contribute to the data integrity and price/performance advantages of Tandem NonStop systems.

Figure 9

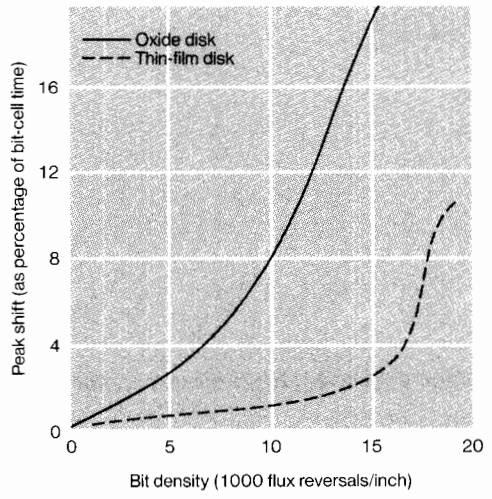


Figure 9.
Peak shift as function of bit density: plated disks versus particulate disks. This graph shows that as bit density increases, the amount of peak shift as a percentage of the bit-cell time increases. The higher the percentage, the higher the read error rate. Plated disks are seen to have a higher bit density for the same desired error rate.

Reference
Taranto, J. 1982. Thin-film Disks Drive Densities to New Highs. *Electronics*, April 21.

Acknowledgment
The author wishes to thank Dick Hodgman and Steve Coleman for reviewing the article and providing valuable feedback.

David S. Ng is the Manager of the Disk Device Development Department. He joined Tandem in 1981 supporting the disk product line. He was project leader for the V8 disk product, and project manager for the XL8 disk product.

Data-encoding Technology Used in the XL8 Storage Facility

The data-encoding technique currently used in Tandem's XL8 storage subsystem is known as 2-7 run-length limited (RLL) code. It provides a 50% improvement in recording density over previous Tandem disk drives.

The preceding article, "Plated Media Technology Used in the XL8 Storage Facility," contains a description of the storage capacity and physical dimensions of the XL8.

This article explains the requirement for a self-clocking recording code used in high-density disk drives and compares the modified frequency modulation (MFM) encoding used in previous disk storage products. It also discusses the 2-7 code, and the implications of the new recording code on testing drive electronics and media defects testing.

Encoding Techniques

Some form of encoding technique is needed to record binary data on the magnetic disk media. Code symbols have been defined as follows:

- A "1" bit represents a flux change.
- A "0" bit represents the absence of a flux change.

The most intuitive encoding technique, and thus the first, was NRZI. A separate signal carrying the clock is used to define a data window (i.e., where the data is to be detected). An example is the Read Data and Read Clock signals on the storage module device (SMD) interface for disk drives.

NRZI, however, soon reaches its limits as data densities increase. This is because of the cumulative effect of the mechanical and electronic components' tolerance and skew of the independent signals.

Self-clocking codes were introduced next. These codes combine clock and data in the same signal. This technique uses a phase-lock loop synchronized to the signal to reconstruct a clock; the clock is then used to recover data.

Run-length limited code is one kind of self-clocking code. RLL code is a coded representation of binary data in which the number of consecutive 0 bits (the run length) is limited by the constraints of the code. This allows the clock to be recovered. By contrast, NRZI code has no run-length limitation; i.e., there can be any number of 0s (nontransitions) before a 1 bit (transition) is encountered.

Table 1.
Representation of NRZI, MFM, and FM codes.

	Data		Coded representation
NRZI	1		1
	0		0
FM	1		11
	0		01
MFM	1		10
	0	followed by 1	00
	0	followed by 0	01

MFM Code

The *Modified Frequency Modulation* (MFM) code is the simplest RLL code. Adjacent magnetic transitions can be as close to each other as 1 bit-cell time and no farther apart than 2 bit-cell time. It is called *modified* frequency modulation because it is modified from a code most commonly found in floppy disk drives called *frequency modulation* (FM). In FM, a clock bit is inserted between every data bit, whether it is a 1 or a 0. MFM removes from FM recording all clock transitions that are adjacent to data 1 bits. The clock transitions fill the gap between 0s. Table 1 shows a coded representation of the three codes discussed in this article.

Note that both FM and MFM have two code symbols (right column) per data bit (left column). Therefore, the detection window for each code symbol is one-half the data-bit-cell time. Why MFM gained popularity over FM is obvious. In FM there can be as many as two flux transitions per data bit, while MFM may have at most one flux transition per bit. FM may have a run of not more than one 0, while MFM may have a run of not less than one 0 and not more than three 0s between 1s.

In magnetic recording, adjacent flux reversals tend to repel each other in the same way like poles of a magnet repel each other. This creates a phenomenon known as *peak shift*.

Peak shift is caused by superposition of signals from adjacent transitions. The peaks of the two signals move away from each other and decrease in amplitude. This introduces error as the read-detection circuitry tries to detect the peaks within the data window. It is, therefore, desirable to have the peaks (derived by differentiating flux transitions) as far apart as possible without losing synchronization with the phase-lock loop. For this reason, MFM is used to gain more recording density over FM. Tandem disk drives 4104, 4105, 4109, 4110, 4114, 4116, and 4120 (V8) all use MFM data-encoding technique.

Figure 1 shows how a binary data stream is recorded on the magnetic medium with the different encoding schemes. Refer to Table 1 to see how NRZ data is coded into flux transitions, represented by 1, and nontransitions, represented by 0. An "encoded write data" pulse corresponds to a "flux transition" at the head. On the read-back, the recorded flux transitions induce the waveform represented by "analog read data." The peaks of this waveform are digitized and decoded back into the NRZ waveforms. The host system can then use the rising edge of the clock signal to retrieve the data.

Figure 1

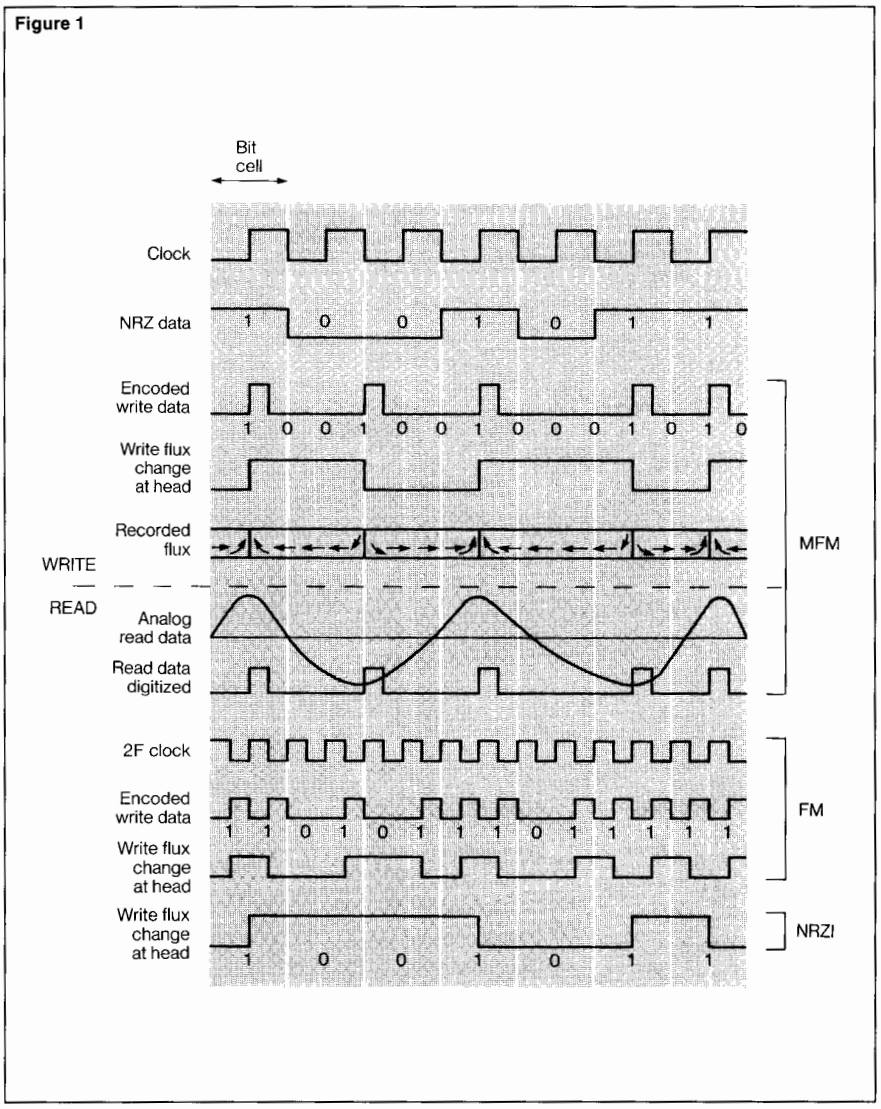


Figure 1.

Comparison of MFM, FM, and NRZI encoding techniques. This figure shows how the same binary data stream of "1001011" is encoded differently using different encoding schemes. The WRITEREAD process is shown in detail for the MFM code. The FM and NRZI codes are included for comparison.

Figure 2

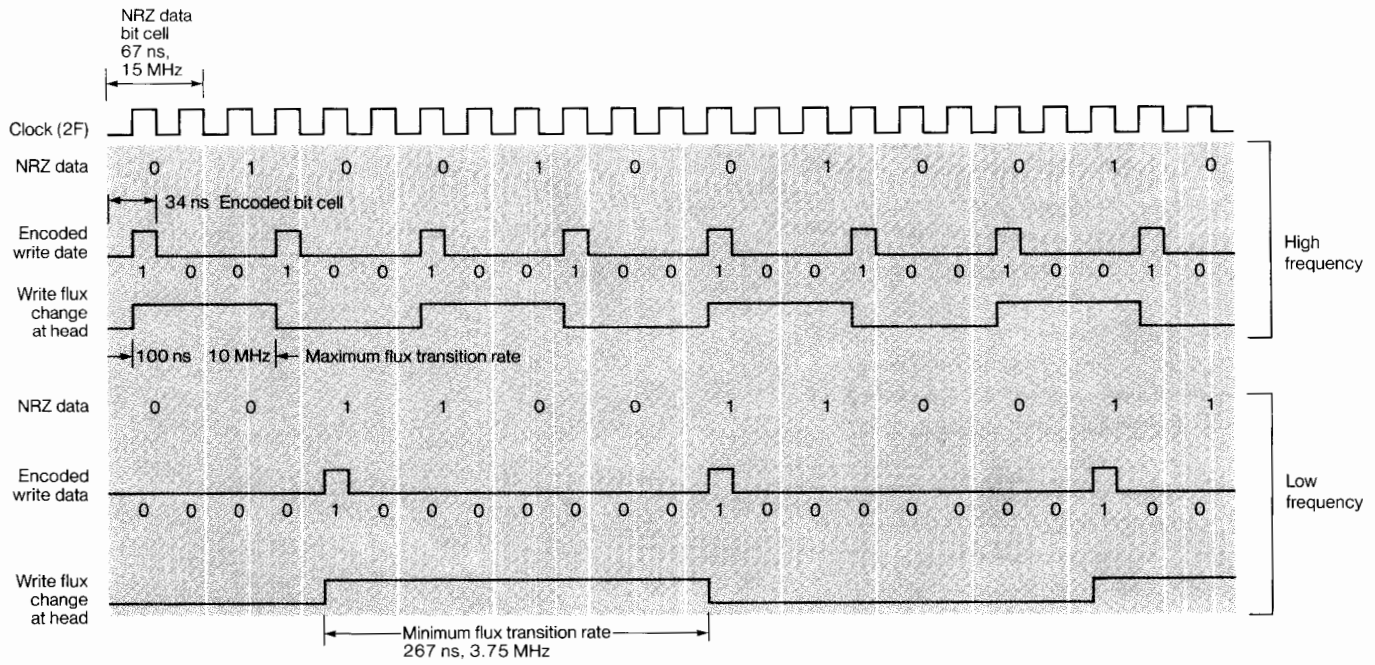


Figure 2.

RLL 2-7 code high-frequency and low-frequency patterns. This figure has the same format as Figure 1. Note that for the high-frequency pattern, there are two nontransitions, 0, between each transition, 1. For the low-frequency pattern, there are seven nontransitions, 0, between each transition, 1.

RLL 2-7 Code

The XL8 disk subsystem uses *RLL 2-7* code, the latest data-encoding technique developed to boost linear recording density. Like FM and MFM, it is also a form of self-clocking code. The first number, 2, refers to the minimum number of consecutive clock cells (half-data bit cell) between flux transitions; the second number, 7, refers to the maximum number of consecutive clock cells without a flux transition. (Within these notation conventions, for example, MFM code can be represented as RLL 1-3.) The advantages of RLL 2-7 code over MFM code include:

- A 50% increase in bits per inch (bpi) is gained with a given flux-transition density. This means that three data bits of information can be derived from the space normally taken by two bits when MFM code is used.
- A lower bandwidth is required for a given data density.
- RLL 2-7 code is optimized for use on present head and disk technologies.

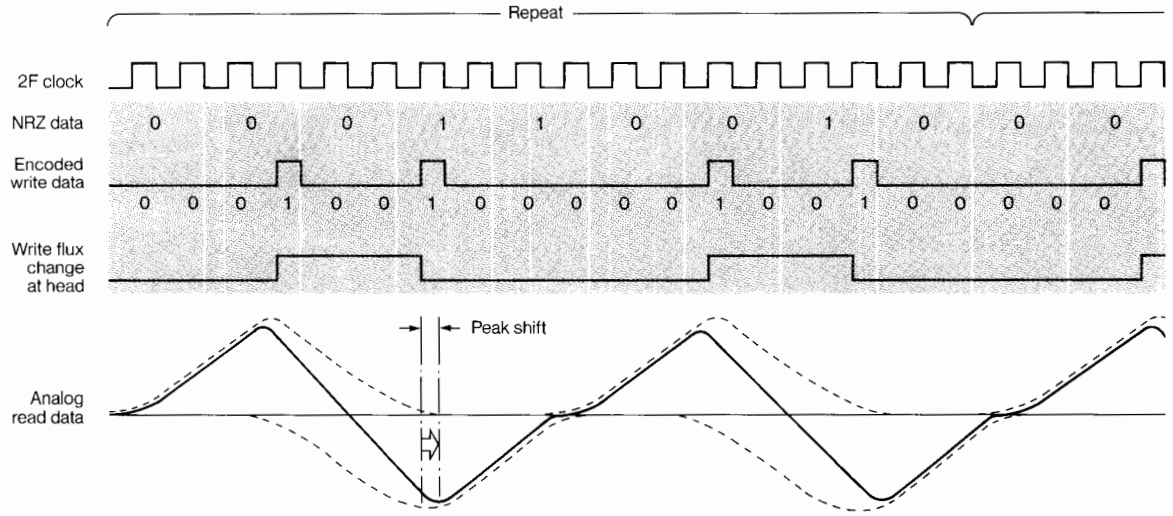
The trade-off is more complicated encoding/decoding and phase-lock loop electronics. XL8 disk drives have a data density of 18,600 bpi, yet the flux transition density is only 12,400 transitions per inch.

Table 2 shows RLL 2-7 code conversion. Note that this is only one of many possible mappings. Note also that the data words (left column) can handle any combination of

Table 2.
RLL 2-7 code conversion table.

Data	Coded representation
10	0100
11	1000
000	000100
010	100100
011	001000
0010	00100100
0011	00001000

Figure 3



binary bit sequences. Moreover, this code conversion does not violate the run-length constraints. This means that each encoded group (right column) must have:

- At least two 0s between the 1s.
- No more than three 0s at the end.
- No more than four 0s at the beginning.

Again, when RLL 2-7 code is compared with MFM code, it is obvious that the flux transitions are now separated by more nontransition bit cells. Therefore, the packing density can be higher. Figure 2 shows the derivation of the high-frequency and low-frequency patterns. The high-frequency pattern has a minimum of two 0s between the 1s; the low-frequency pattern has a maximum of seven 0s between the 1s.

Each bit-cell time is 67 ns for a bit rate of 15 MHz. Yet with the high frequency pattern, flux transitions occur at a maximum rate of 10 MHz, with each magnetic domain lasting 100 ns.

As shown in the conversion table, it takes two encoded bit times (right column) to represent one data bit (left column). On the XL8, the 9-inch diameter disk spindle rotates at 3070 rpm; thus, each encoded bit time is 33.3 ns wide. Since there must be at least two 0s (or nontransitions) between the 1s, the maximum rate at which the 1s (or transitions) occur is once every three bit times, or every 100 ns. Although the maximum flux transition frequency is 10 MHz, this represents a binary data transfer rate of 15 MHz.

Figure 3 shows how a binary data pattern is converted into RLL 2-7 code and recorded onto the magnetic media. The pattern used is the peak shift pattern, chosen to measure worst-case peak shift.

Figure 3.
Peak shift pattern for RLL 2-7 encoding. The peak shift pattern can be used to test data error rate induced by peak shift. It consists of two 1s with minimal separation (two 0s) to maximize the peak shift effect. These groups of two 1s are separated by as many as five 0s to minimize interaction between the groups.

Testing Drives with RLL 2-7 Code

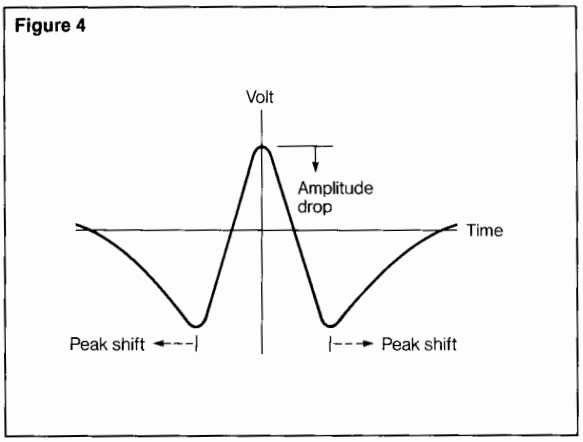
Since binary test patterns now go through a different conversion table than that used for MFM code, a different set of test patterns must be generated for media defect detection as well as other evaluation tests.

Media Defect Testing: Format Patterns

Winchester disk drive manufacturers use both analog and digital methods to test the drive for media defects as part of the production process.

Figure 4.

Three-flux transition pattern used for formatting. With two transitions written as closely as possible, the center transition's amplitude is seen to have dropped. This is used to detect marginal recording cells on the media.



The analog method detects defects by examining the analog data signals before they are digitized and decoded. Missing bits, extra bits, and amplitude modulation criteria are used.

The digital method involves writing a certain binary data pattern to the drive through the interface and reading it back for comparison. At Tandem, sectors containing defects are spared during the format process so that customer data will not be written on them. We require the drive manufacturer to supply a list of defect locations detected at the factory. Our formatting process uses the vendor list as a base and adds any other defective spots found during the format process.

The patterns used during the format process are chosen deliberately to improve defect detection. These patterns are designed to force worst-case amplitude modulation. This can be achieved by superposition of signals from adjacent flux transitions. When three flux changes are written with equal spacing and are close enough for interaction, the outside signals appear to have their peaks shifted away while the center one has a diminished amplitude. This middle transition is used for testing marginal defects. (See Figure 4.)

Ideally, this group of three transitions should be written repetitively with minimal spacing between each transition; the groups should be separated to minimize interaction between the groups.

In order to test for every possible defect location, the middle bit must be made to scan. The task is to pick patterns that cause this middle bit to scan across the track. In MFM code, four patterns are necessary to complete the scan. The binary pattern of 01110111... is encoded into 0010101000101010000.... The two underscored bits are tested.

RLL 2-7 requires over 12 passes (i.e., 12 different patterns) for complete coverage. These patterns are derived and used during new product development to evaluate media quality and defect characteristics. It may be prohibitively expensive to use all possible test patterns during the manufacturing process.

The vendor's analog tests usually find over 85% of the media defects. The digital tests used by the vendor and Tandem find another 10%. Possibly 5% of the marginal defects are not detected before the drive is shipped. These are the locations that may fail once per ten reads or even once per 100 reads. Defects that cause errors in normal operation do not cause problems for users in the great majority of cases.

Defects tend to be limited in length, and the error correcting code (ECC) built into the disk controller corrects most errors caused by defects, as it does other errors. They are reported as corrected data errors and their correction is accomplished without any noticeable impact on performance or data access time. Correctable data errors that occur repeatedly on the same physical location can easily be mapped out by a PUP command.

Test Patterns

Three other tests are sensitive to data patterns. These patterns are used to test the read/write electronics of the disk drive.

Overwrite Test Pattern. Overwriting is a test to evaluate the drive's ability to erase old data patterns with new ones; i.e., it is a test to detect the residue of the old pattern after the new pattern is written. It can be done digitally or with a spectrum analyzer.

Typically, a low-density signal permits higher write current before saturation occurs, and vice versa. Most disk drives use only one

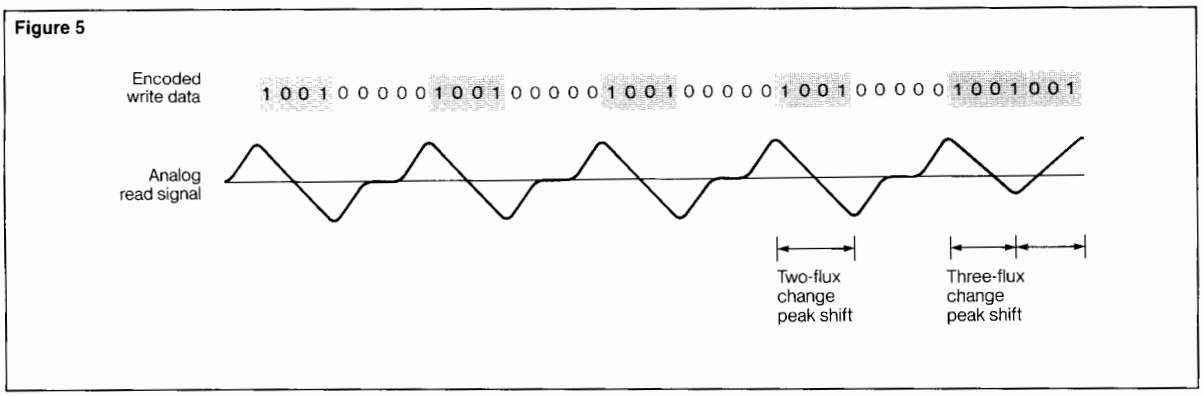


Figure 5.
Read reliability pattern. The read reliability pattern is used to test the design of the read phase-lock oscillator (PLO). It consists of four groups of 1 0 0 1 separated by five 0s. This causes the phase-lock oscillator to adjust in one direction. Then a group of 1 0 0 1 0 0 1 is inserted to throw the PLO off balance as it tries to adjust to a different amount of peak shift.

level of write current to simplify design. The write current is optimized at just above the knee of the saturation curve for the highest density (inner track, high-density signal). This means that the lowest density (outer track, low-density signal) is undersaturated. The test involves writing with the high-frequency pattern twice on a track and then writing the same track again with the low-frequency pattern. The track is then read and verified. Both patterns shown in Figure 2 are used for RLL 2-7 encoding. With MFM, the high-frequency patterns are 1111... or 0000...; the low-frequency pattern is 1010....

Track Misregistration Pattern. The same high- and low-frequency patterns can also be used to run the track misregistration test. This tests the "cross talk" between the tracks. Typically, the two patterns are written alternately on the innermost tracks at one temperature extreme, and then read back at the other extreme. If an error is detected, it may indicate a thermal expansion problem on the base plate or actuator, or a tracking error on the servo system.

Read Reliability Pattern. This test is designed for testing the phase-lock loop's ability to respond to changes in the relationship between the data and the clock information. As with all self-clocking codes, the signal read from the disk carries both the clock and data information. The data window recovered from the signal itself is used to check for the occurrence of a flux transition. Due to the superposition of signals of adjacent flux transitions, however, the peaks tend to shift away from the center of the data window. Depending on the flux transitions that are written, different patterns cause varying amounts of peak shift. Since the clock is also derived from the same signal, the effect is compounded. In order to handle the varying data peak and data window relationship, the design of the phase-lock loop has the effect of a flywheel so that occasional irregularities do not cause large corrections.

The ideal read reliability pattern performs two functions:

1. It introduces maximum peak shift in the read data.
2. It causes the phase-lock loop to make successive corrections in one direction, and then switch direction so that the window is at the maximum offset relative to the data. This makes the peak shift appear to be even worse.

The maximum peak shift is obtained by writing two flux changes as close to each other as the encoding scheme permits. This two-flux change pattern as a group is then written several times consecutively. To minimize intergroup interaction, as many nontransition cells are written between the groups as the encoding scheme permits. A three-flux change pattern (used for formatting) is then inserted to throw the phase-lock loop to the opposite direction. The ratio of two-flux transition patterns to three-flux transition patterns should be determined by experimenting with the individual drive. Figure 5 uses a 4:1 ratio as an example.

Table 3.
RLL 2-7 read reliability pattern.

Encoded read reliability pattern	100100	000100	1000	00100100		000100	1000	00100100	1000	0010010	000100
Decoded binary pattern	010	000	11	0010		000	11	0010	11	0010	000
Binary	0100	0011	0010	0001	1001	0110	0100	00....			
Hex data	4	3	2	1	9	6	4				

With MFM code, the maximum peak shift pattern is "110110110," etc., and the reversal pattern is "01110." Combining the two patterns in a 4:1 ratio yields "1101101101101110," etc. Converting this pattern to hexadecimal yields the famous "DB6E."

Table 3 shows how the RLL 2-7 read reliability pattern in Figure 5 is decoded into hex data. Since we cannot find a pattern that uses the maximum seven 0s between the 1s, we have settled for five. Refer to Table 2; remember that there is a minimum of two 0s between the 1s.

The first line is the encoded bit pattern as written on the disk. Note that there are four two-flux change groups followed by a three-flux change group. Each group is separated by five 0s. Spaces are inserted to show conversion relationship with the second line. The third line regroups the second line into groups of four binary bits. The fourth line is the hexadecimal representation of the binary pattern in line three. Thus, the read reliability pattern for this particular conversion table is "432196" hex.

Conclusion

The RLL 2-7 encoding technique leverages off existing head and media technology to provide a 50% improvement in recording density. Users gain the benefit of a faster data transfer rate and smaller physical volume, with the reliability of proven recording head-media technology. New test patterns are derived and applied during the product development phase; they have proven that good electrical signal properties are not sacrificed at the expense of gaining recording density. The principles presented here can be adapted to evaluate and test future disk products with other self-clocking coding schemes.

Acknowledgment
The author wishes to thank Dick Hodgman and Steve Coleman for reviewing the article and providing valuable feedback.

David S. Ng, Manager of the Disk Device Development Department, wrote this article, as well as the preceding article, "Plated Media Technology Used in the XL8 Storage Facility."

Data-window Phase-margin Analysis

Tandem uses several criteria to select the best possible disk drive vendors. Besides considering financial status, long-term viability, and product suitability, Tandem also undertakes plant inspections, diagnostic testing, and visual drive inspections to narrow the number of vendors to three candidates. Usually six to ten samples of the drive model are obtained from each of the three vendors and subjected to diagnostic tests, environmental tests, burn-in tests (to identify infant mortality problems), and life tests to ensure that the unit will meet Tandem's needs.

One performance parameter verified is the manufacturer's stated error rates. "Errors" (especially recoverable errors) should not be confused with "disk failures." Errors are generally not catastrophic, but they waste valuable processing time. Measuring this parameter not only identifies a poorly designed drive, but helps differentiate between two well-designed drives.

Predicted and measured recoverable error rates for a well-built and well-designed Winchester or Whitney drive are one in ten gigabits (10,000,000,000); nonrecoverable error rates for these drives are one in one terabit (1,000,000,000,000).

Previously, Tandem accepted vendor claims at face value. It is costly and time consuming to quantify the actual soft error rate by normal methods:

- At current average access times (20 ms) and current average block lengths (256 Kbits) it takes 11 hours to run a single sample of 10E10 bits.
- Running a large enough number of samples (e.g., 25) to reduce stochastic error to a comfortable level would take 265 hours for each drive.
- All samples submitted would need testing to ensure that the sample was representative. Even using many processors, such testing would overwhelm laboratory resources.

Tandem needed a way of accelerating testing and extrapolating data without abrogating accuracy. *Data-window phase-margin analysis* provides the desired result without resorting to a full worst-case analysis or running masses of data.

Data Windows and Bitshift

The *data window* is that space (or period) in time in which the data bit must appear to be recognized as correct data. (See Figure 1 on the following page.) Errors occur when a bit has moved into the next data window in a process known as *bitshift*. Bitshift is defined as the time that the bit in question deviates from its preferred position, i.e., the center of its data window. Knowing how well a disk drive manages its overall bitshift provides a good measure of the true capabilities of the drive. It is analogous to comparing marksmen by measuring the spread of their shot.

Figure 1

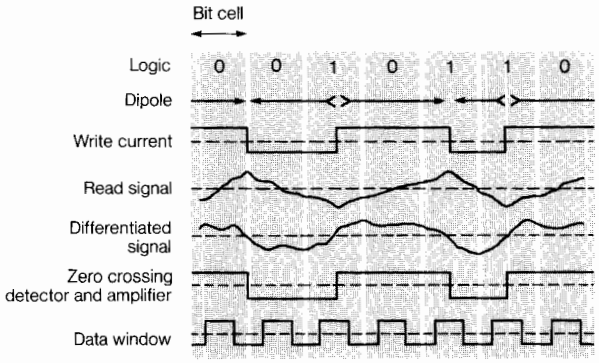


Figure 1.
MFM recording technique.

Figure 2

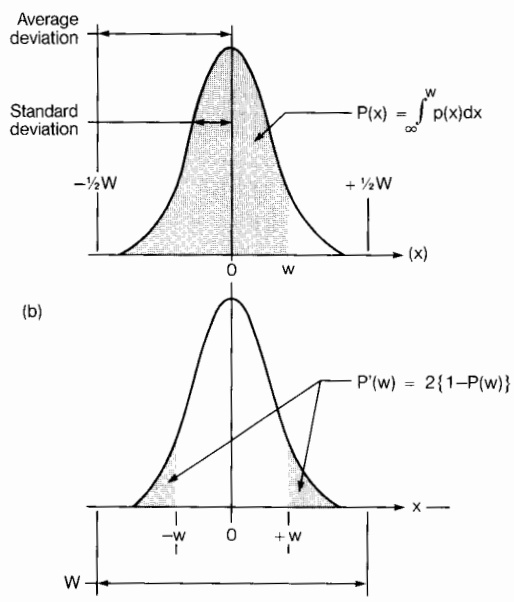


Figure 2
The probability density distribution curve for bitshift of random data patterns. (a) The standard probability function. (b) The portion that falls outside some given period from the center of the curve.

Bitshift is caused by a combination of many factors including:

- Delays in the writing and reading channels.
- Magnetic and electronic noise.
- Adjacent domain interference.

Recording Techniques

Compensation circuits are used to correct for read/write delays. The effect most difficult to manage is bitshift caused by adjacent-domain interference. This occurs when two closely juxtaposed magnetic domains on the disk interfere with each other.

Magnetic recording is done by creating small permanent magnets called domains on the coating material covering the surface of the disk. The magnetic polarity of these domains can be reversed by the influence of external magnetic forces. Just as permanent magnets influence each other when placed close together, the domains in close proximity influence each other's magnetic vector, causing the peaks of the pulses that are reproduced through the read head to move closer together or farther apart.

Modified Frequency Modulation

Figure 1 illustrates the modified frequency modulation (MFM) recording technique. The following rules apply to this type of recording:

- *ones*: A reversal of magnetic flux direction occurs in the middle of the bit cell.
- *zeros*: No reversal of flux occurs in the middle of a bit cell, but a reversal is to occur at the leading edge of the bit cell, whenever a zero follows a zero.

Note that the data window always equals half the bit cell in MFM recording.

Plotting Curves

The following method obtains the typical normal curve shown in Figure 2:

1. Run a fixed sample of bits.
2. Superimpose each one on its data window.
3. Use the center point of the generated period as the zero point.

Each pulse is shifted either positively or negatively, with respect to that zero point. If the sample is large enough and the shifting is random, the distribution of those bits is Gaussian[1] about that zero point and fits a normal curve as generated by that sample.

Conversely, the probability of finding a bit shifted a specific amount from the zero point in any similar sample of data from that particular disk drive obeys the same normal curve. (At this point the concern is not with the normal curve per se, but rather in how close its skirts come to the window limits.) Phase-margin analysis determines the data window in which no pulses appeared, after some fixed predetermined quantity of data has been run.

Typical Data-error Curve

Measuring P'(x) for values of x from 0 to +W where +W is the positive limit of the data window, results in the curve shown in Figure 3. By calculating the standard deviation (S), it is possible to use the Gaussian tables to extrapolate our results to any sample size (e.g., 10E10). The standard deviation is defined as the manner in which the function P'(x) varies about the mean deviation. In this case, the mean deviation is the zero point and the standard deviation is the shape of the curve.

$$S = \frac{+w(\%)}{g}$$

where g is the Gaussian value for P'(x) when $x = |w|$

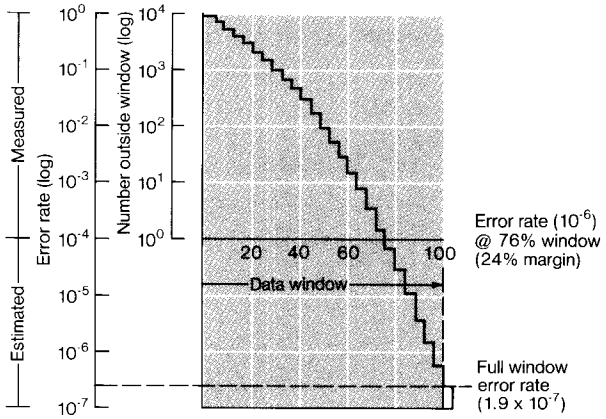
$$p(x) = \frac{1}{S\sqrt{2\pi}} e^{-1/2\left(\frac{x-\bar{x}}{s}\right)^2}$$

where $\bar{x}$ is the mean deviation and s is the standard deviation. A good example of this is the failure of a piece of electronic gear where the mean time between failure (MTBF) is the mean deviation.

The curve is physically plotted as follows:

1. Measure P'(x) at 1-ns intervals from 0 by artificially narrowing the data window about its theoretical center and counting the number of bits that fall outside the narrowed data window.
2. Run 10E5 bits and plot P'(x) from its maximum (10E5 at data window = 0) to its minimum (P'(x) = 0). The resulting curve is similar to that shown in Figure 3.

Figure 3



3. Calculate the standard deviation for several values to ensure that the distribution is Gaussian.
4. Use the Gaussian tables to calculate P'(x) for values of x from the measured limit (w) to the data windows' limit (W).

Note that this technique is only accurate for Gaussian distributions about the assumed mean deviation. If the curve is distorted (i.e., it has one or two shoulders or more than one peak), window sliding (delaying the data window with respect to the data) shows the true mean deviation. If the mean deviation is not in the center of the data window, presumably something is radically wrong with the design of the drive (e.g., timing circuits, phase-locked oscillators [PLOs], write compensation, or read compensation, etc.).

If the standard deviation is not a constant, there could be something wrong with the sample (e.g., pattern sensitivity) or with the drive (e.g., read/write head, write current, media, synchronous noise, etc.).

Figure 4 on the following page contains some of the equations describing the bitshift manifestation.

Figure 3.
Typical data-error curve. This curve is the composite of P'(x) as shown in Figure 2(b). It is a count of bits falling outside the data window from W = 0 to 77% of Max. W. The curve is then extrapolated to W = 100% of Max. W.

[1]The Gaussian distribution is known as the normal distribution by statisticians. It is a bell-shaped curve showing a distribution of probability associated with different values of a variate.

Figure 4

Assume Gaussian distribution:

(a) The probability[1] of finding any bit displaced from its designated position by time t:

$$P(t) = 1/2 \sum N\{\exp[-(t-b_d)^2/2b_n^2] + \exp[-(t+b_d)^2/2b_n^2]\}$$

where N = # of bits (sample size)
b_n = noise bitshift
b_d = domain interference bitshift
t = time displaced from true position

(b) The intrinsic error rate, E, is the probability of finding a bit outside the data window or:

$$E = \int_{-\infty}^{-w} P(x)dx + \int_{w}^{\infty} P(x)dx = 2(1-P(x)) = P'(x)$$

(c) The mean deviation[2], $\bar{x}$, is:

$$\bar{x} = \sum_{i=1}^{n} \frac{x_i}{n}$$

(d) The standard deviation, S, is:

$$S = \sum_{i=1}^{n} [(x_i-\bar{x})^2/(n-1)]^{1/2}$$

[1]The probability and error rate formulas are discussed in "Effect of Bitshift Distribution," Katz, 1979, *IEEE Transactions*, Vol. 15.

[2]The mean deviation and standard deviation are referenced in the *Standard Handbook for Mechanical Engineers*, 8th Edition, Beaumeister, pp. 17-19.

Figure 4.
Equations describing the bitshift manifestation. Equation (a) is the density probability for bitshift due to noise and domain interference. Equation (b) describes the density probability for bits falling outside a given window. Equation (c) is the mean deviation, which in this particular case should coincide with the center of the data window. Equation (d) is the standard deviation that describes the density curve.

Figure 5

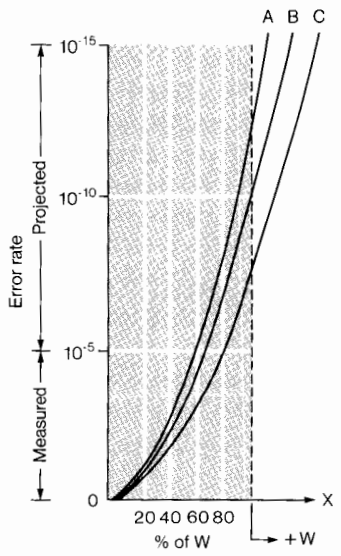


Figure 5.
XL8 candidates A, B, and C.

Recent Study Results

Figure 5 shows the results of the recent study to select the optimum drive for the XL8 disk storage facility. The study compared the estimated percentage of data window remaining at 10E10 bits of random data and calculated the intrinsic error rate.

This plot (Figure 5) was inverted from the typical plot shown in Figure 3 for ease of automation. Diagnostics indicated that two of the candidates were equal performers. Phase-margin analysis verified that the intrinsic error rate was better than the manufacturer claimed. It also indicated that one of the drives had a greater margin of data window remaining at the 10E10 level and, therefore, had a greater probability of errorless operation.

The vendor candidates showed the following intrinsic error rates:

- Candidate A had approximately one in 5 x 10E12 bits.
- Candidate B had one in 10E10 plus bits.
- Candidate C, with an error rate of one in 5 x 10E7 bits, did not meet Tandem's requirements.

Conclusion

Future studies at Tandem will use window sliding to determine the extent to which the bitshift is symmetric about the center of the data window. This will help determine the effectiveness of the read/write compensation circuits, and the quality of the media and its relationship to the write current.

Note that data-window phase-margin analysis is not used to determine whether a drive is performing correctly in its system environment. Diagnostic routines do that much more readily. It does, however, help ascertain which drive has the greatest potential for reliable operation, and is one of several investigative methods responsible for the high reliability of Tandem's disk drives.

Alan Painter is a Manager in Hardware Engineering Peripherals. He has been with Tandem since October 1982 and has 30 years experience in the design of disk and tape drives.

Hoa Pham is a mechanical engineer in the peripherals department and has been with Tandem since 1983. Hoa is a member of the Mechanical Engineering Team led by Joerg Ferchau under Al Painter. The team recently won the Hanover Fair's "Gute Industrieform" award for its work on the XL8.

Herb Thomas is a technician in the peripherals department. He provided valuable technical assistance in this study.

This column summarizes Tandem product announcements for the fourth calendar quarter of 1985. For ease of reference, new products are listed in alphabetical order.

B20 Release—NonStop System Software

The B20 release of the NonStop system software, available November 1985, incorporates a significant number of product enhancements, new products, and bug fixes. Tandem recommends that all customers install it.

The new products column in the February 1986 issue of the *Tandem Systems Review* discussed the following B20 release products:

- C compiler.
- COBOL and FORTRAN separate run-time libraries.
- TACL, a flexible command interpreter.
- Information Management Technology (IMT) products, PS MAIL™ for 6530 terminals, PS MAIL for 3270 terminals, and PS TEXT EDIT.
- TAL compiler enhancements.

This column describes the following new B20 products:

- Single-ported Communication Interface Unit for the 6100 Communications Subsystem.
- SAFE-T-NET™ encryption subsystem.

DP2 and TMF

DP2 and the network Transaction Monitoring Facility (TMF) are generally available with this release. The June 1985 issue of the *Tandem Systems Review* contains articles describing the benefits of DP2. By upgrading to the B20 release, customers get autorollback, downed volume reintegration, increased reliability, and DP2 support.

DP1 is functionally stabilized as of the B20 release. DP1 will be included in the B-series releases, but will not be shipped with C-series releases.

TMDS

The B20 release includes two new subsystems of the Tandem Maintenance and Diagnostic System (TMDS), which provides system support for on-line diagnostics. TMDS was first released in the B00 release with the FOX™ diagnostic. (See "Introducing TMDS, Tandem's New On-line Diagnostic System," in the June 1985 issue of the *Tandem Systems Review*.) The B20 release contains TMDS subsystems for disk and tape diagnostics.

Labeled Tapes
Labeled tape handling is available in the B20 release, but distribution will be limited until the B30 release while the product is tested at customer sites. Contact a Tandem systems analyst for more information. The B20 release provides the following features for processing labeled tapes:

- Labeled tapes can be accessed from COBOL and TAL applications.
- ANSI standard (X3.27-1978) and IBM standard (GC26-3795-3) labels are supported.
- Nine-track tape is supported in three different densities: 800 (NRZ), 1600 (PE), and 6250 (GCR) bits per inch (bpi).
- Each tape file is identified to the GUARDIAN 90 operating system by a unique logical file name; a set of attributes describing the file is collected in a new structure.
- The tape process recognizes the tape mount and reports the mount information to a process that performs Automatic Volume Recognition (AVR).
- A tape management utility program (TAPECOM) provides the user interface for tape-related operations.

INSPECT Symbolic Debugger
INSPECT supports a new SOURCE command that can be used to display the source program statement(s) corresponding to a code location.

6100 Communications Subsystem
The X.21 call-control interface is supported by new 6100 Communications Subsystem (CSS) products: Line Interface Module 6129-7 (X.21 LIM) and Line Interface Unit (X.21 LIU). The 6100 ADCCP protocol module allows an application to set up an X.21 circuit-switched connection and communicate over the circuit using the ADCCP bit-synchronous communication protocols.

TCP
A-series and B10 releases of the GUARDIAN operating system included both the old and the new TCPs to facilitate migration to the new version. The new TCP, PATHTCP2 introduced in A06, offers significant performance improvement by using extended data segments and reducing disk I/O operations in the TCP. Many sites have converted in the last 18 months with no significant problems. Because of our customers' positive experience with the newly designed TCPs, Tandem has split the B20 PATHWAY transaction processing system into two products: a NonStop version (T9153 PATHWAY) and a NonStop 1+ version (T9103 PATHWAY). Only NonStop systems will support PATHTCP2 and SCOBOLX (which replaces SCOBOL); only NonStop 1+ systems will support PATHTCP and SCOBOL.

In a future release, ENFORM and DDL will also split into two products—a NonStop systems version and a NonStop 1+ systems version.

For additional information, see "A New Design for the PATHWAY TCP" (*Tandem Journal*, Spring 1984) and "The PATHWAY TCP: Performance and Tuning" (*Tandem Systems Review*, February 1985).

Product Overview

Tandem has recently released the following new or enhanced products:

- 5110/5114 tape drives for NonStop EXT systems.
- 5130/31 tape subsystem.
- 6100 Communications Subsystem enhancements.
- 6535/36/37 Ergonomic Terminals.
- DYNAMITE™ workstation enhancements.
- PC LINK™ Workstation Software Site Licenses.
- SAFE-T-NET cryptographic device.
- SAFEGUARD system protection software (planned for release in the spring of 1986).
- T-TEXT™ support for local printers.
- XL8 disk storage facility.

Literature is available for many of these products from Tandem sales representatives. The "SAFE™ Integrated Security Products" brochure describes SAFE-T-NET and SAFEGUARD. The "Disk Drives" brochure describes both the V8 and XL8 disk storage facilities. Information sheets are available for the Communications Control Subsystem (CCS) and National Language Support enhancements for the DYNAMITE workstations. A data sheet is available for the 5130/31 Tape Subsystem.

5110/5114 Tape Drives for NonStop EXT Systems

Two stand-alone tape drives are available for the NonStop EXT system: the 5110 and 5114. Like the NonStop EXT, these tape drives do not require a computer-room environment. Both devices are nine-track reel-to-reel tape drives; the 5110 tape drive operates at 45 inches per second (ips) and the 5114, at 125 ips. Each drive comes complete with cabling, a stand-alone cabinet, and a patch panel for the NonStop EXT system.

The 5110 is a low-cost utility drive operating at 45 ips. The device provides both 800 bits per inch (bpi) NRZI and 1600 bpi phase-encoded formats.

The 5114 tape drive operates at 125 ips, providing both 800 bpi NRZI and 1600 bpi phase-encoding formats.

Storage capacity of these tape drives varies depending on the recording density and block size that are selected. At 1600 bpi, with a maximum recommended block size of 8 Kbytes, 40.6 Mbytes can be stored on a 2400-foot tape. At 800 bpi, with a maximum recommended block size of 4 Kbytes, 20.6 Mbytes of data can be stored on a 2400-foot tape.

There is an option for NonStop EXT packaged systems (including NonStop EXT/TXP package upgrades) that allows the customer to substitute the 125 ips drive for the standard 45 ips drive.

5130/31 Tape Subsystem

The 5130/5131 is a mainframe-class tape subsystem that provides fast and efficient tape operations. A tape speed of 200 ips, coupled with recording densities of 6250 bpi Group Coded Recording (GCR) and 1600 bpi Phase Encoding (PE), provides fast, efficient backup of large data bases. Along with high performance, the 5130/31 incorporates user-friendly features such as power windows and tape autoloading. The 5130 tape subsystem includes a tape transport, a formatter, a control unit, and the cables required to install a single-drive subsystem. Up to three additional 5131 tape transports can be attached to a 5130 to configure a maximum, four-drive subsystem. The 5130/31 transfers data at a maximum rate of 1.25 Mbytes per second.

The 5130/31 can read and write ANSI-compatible PE and GCR tapes. Using GCR, the tape subsystem provides up to 180 Mbytes of storage on a single 2400-foot reel of tape. Additional features such as tape quality monitoring, an innovative tape transport design, and diagnostic functions ensure data reliability.

6100 CSS Enhancements

New protocols for the 6100 Communications Subsystem (6100 CSS) are developed for use on the 6100 CSS under the CP6100 Communications Access Process. Any customer can request a protocol by having a Tandem analyst or sales representative fill out a Protocol Request Form. However, Tandem reserves the right to decide whether or not to approve the protocols.

Single-ported Communication Interface Unit. Many of our customers have complained about the number of channel addresses used by a single 6100 Communications Subsystem. Previously, the 6100 CSS required 64 of the 256 available channel addresses. The Single-ported Communication Interface Unit (CIU) solves this problem by reducing the number of required channel addresses to 32.

UTS-40 Supervisor Protocol. This protocol provides software for the Sperry Univac Universal Terminal System 40 (UTS-40) Multipoint Supervisor protocol line task. The UTS-40 Supervisor provides the ability to control multiple terminals on a synchronous communications line operating under the Univac UTS-40 polling protocol. Terminals supported are those that conform to the protocol described in the Sperry Univac UTS-40 Single Station System Reference (Univac number UP-9143-B).

UTS-40 Tributary Protocol. Provides software for the Sperry Univac UTS-40 Tributary multipoint protocol line task. The UTS-40 Tributary allows the Tandem host system to look like one or more secondary stations to the supervisor. This means that users can write an application simulating multiple terminals associated with a single Remote Identifier (RID), on a synchronous line operating under the UTS-40 polling protocol. Terminals simulated are those that conform to the protocol described in the Sperry Univac UTS-40 Single Station System Reference (Univac number UP-9143-B).

V.35 Line Interface Unit. The new V.35 LIU currently supports CSSADCCP and X.25 software. The LIU consists of one Communications Line Interface Processor (CLIP) and one V.35 electrical interface, Line Interface Module (LIM). This LIM uses a 25-pin connector rather than the 34-pin connector described in the V.35 standard. Customers must supply a special cable. A wiring diagram is included with each LIM.

6535/36/37 Ergonomic Terminals

Tandem's three new ergonomic terminals offer all the functionality of 653X models, plus new low-profile keyboards. These terminals take up less desk space and meet European ergonomic requirements. New models are the 6535 with a 15-inch diagonal screen, the 6536 with a 12-inch diagonal screen, and the 6537 with a 9-inch diagonal screen. These models are compatible with all existing 6530 terminal options, except T-TEXT word processing capability, and with all system and application software.

The new terminals maintain the advanced ergonomic features of the original 653X family, including detachable keyboards, nonglare screens with green phosphor characters, and low-contrast colors to ease eye strain. Tilt/swivel and a 6-foot electronics-to-monitor cable set are standard on all models. With the addition of the low-profile keyboard, the new models meet the German DIN ergonomic standard, a set of design specifications for operator comfort required for selling terminals in many European countries, including Germany and the Scandinavian countries.

The 6535 typewriter-style keyboard is identical in size to the DYNAMITE workstation keyboard, expanding application opportunities in limited-space environments. Similar in layout to the 6530 keyboard, the new keyboard has simplified cursor key positioning in relation to the alpha and numeric keypads. The keyboard maintains the 16 program function keys and two-position tilt adjustment of the 6530 keyboard. Keyboards are available to match the international language character sets.

DYNAMITE Workstation Enhancements

There are five enhancements for the DYNAMITE workstation including two new option cards, two new communications products, and national language keyboards and software. Today's new DYNAMITE products and other recent product additions have greatly expanded the original DYNAMITE offering of just six months ago. There are now hard disk models, color models, a half dozen communications products, and also a half dozen feature options.

AM6520 Communications Software. The AM6520 Communications Software provides byte-synchronous multipoint connection of DYNAMITE workstations to AM6520 and to 6820 Terminal Cluster Concentrators (TCCs). This product consists of the AMT6530 terminal emulator and the AM-IXF file transfer software on diskette. The DYNAMITE workstations can be mixed with 653X terminals currently being used.

Communications Control System (CCS). CCS is a unique product which helps customers write MS-DOS applications that can communicate interactively with a host computer system or other device. It consists of an asynchronous communications driver and a set of C language functions that can be linked with the application to transfer streams of characters to and from the DYNAMITE workstation.

Graphics-combo Card. This product provides a comprehensive set of features on one card. Included are graphics that support color or monochrome IBM PC-compatible graphics (320x200, 640x200), high-resolution graphics (800x300), an IBM PC-compatible communications port, a parallel printer port, and a real-time clock with battery backup.

Multifunction Card. The Multifunction Card is for applications that don't require graphics, but require a parallel printer port, IBM PC-compatible asynchronous communications port, and/or real-time clock. All three are included on one card.

National Language Support. This support includes keyboards and character sets for the most frequently required languages, and new international versions of MS-DOS. The national characters are supported both in the 653X mode and the MS-DOS (IBM PC) mode. The character sets supported in the first release are: French, German/Austrian, Swedish/Finnish, Danish, Norwegian, Spanish, and U.K.

PC LINK Workstation Software Site Licenses
Tandem offers a variety of PC software site licenses to meet the needs of virtually all of our customers. Currently Tandem's PC6530 product is covered by this program. PC6530 workstation software, part of the PC LINK product group, includes 6530 terminal emulation and Information Xchange Facility workstation software for IBM PCs and other workstations compatible with the PC.

The *Corporate License* gives the customer the right to make unlimited copies of specific software for workstations connected to any of the customer's Tandem systems. The customer receives one copy of the software, including documentation, from which to make additional copies.

The *System License* gives the customer the right to make unlimited copies of specific software for workstations connected to a specific Tandem system. The customer receives one copy of the software, including documentation, from which to make additional copies.

Tandem also offers an update option that provides the customer with the right to make copies of any new releases of the software. The update option covers new releases issued during a one-year period from the date of the License Order for the product.

For customers who purchase Corporate or System licenses, but would rather not make their own duplicates, Tandem provides copies of the software and documentation at a special rate.

SAFE-T-NET Encryption Subsystem
SAFE-T-NET cryptographic device is a channel-attached peripheral device that performs cryptographic functions with Tandem systems. The product provides data encryption, message authentication, and the capability to change the master key on-line.

SAFE-T-NET utility functions include encryption of SNAX terminal sessions on IBM 3270 and PCs with cryptographic capabilities, and general-purpose encryption via a file-system interface. The device complies with the U.S. National Bureau of Standards' Data Encryption Standard (DES).

The on-line master key change facility is implemented by a patented mechanism. This feature maximizes availability and promotes sound security practices by allowing security administrators to change encryption keys without affecting system availability.

SAFEGUARD System Protection Software
SAFEGUARD distributed system security software, available with the B30 NonStop systems software release in the spring of 1986, provides users of Tandem distributed networks with mainframe-level protection mechanisms that can be controlled from a single interface.

SAFEGUARD software authenticates the identity of users who attempt to access the network. Any logon attempts, whether successful or not, can be recorded. Once users are authenticated, SAFEGUARD protects all system resources, allowing access only to authorized users.

System managers and security administrators can also control access to any other shared network resource, including terminals, processes, printers, encryption devices, tape drives, and communication lines. The authorization mechanism allows the security administrator to specifically define a list of users (both local and network) that have access to any of these resources.

The ability to audit activities that involve shared resources is important in any security system. SAFEGUARD software allows security administrators to selectively record attempts to access any data file or shared network resource.

See the accompanying article, "Distributed System Protection with SAFEGUARD," for more information.

T-TEXT Support for Local Printers
Effective with the B20 release of T-TEXT software, 6530, 6531, or 6532 terminals with T-TEXT word processing capability installed will be able to support locally attached 5530 Letter Quality Printers with full T-TEXT formatting capability as a system-addressable printer through the new 6LAT Local Printer Interface Option.

The printer interface option is used to cable a serial printer, including Tandem's 5520, 5530, and 554X models, directly to a 653X terminal. The printer can be configured as a local screen printer or as a separately addressable printer accessed via the AM6520 software.

In the local screen print configuration, the PRINT key on the 653X and T-TEXT keyboards can be used to print the contents of the host 653X screen. In the separately addressable configuration, the printer is treated as a separate subdevice on an AM6520 communications line, so any user on the host computer system can access the printer through the system software such as SPOOLER, T-TEXT/TFORM, and FUP.

XL8 Disk Storage Facility
Designed specifically for high volume on-line transaction processing, the XL8 disk storage facility provides exceptional storage capacity for NonStop systems. Tandem has pioneered the most advanced VLSI and thin-film media technology in the industry to offer this capacity at a significantly lower cost per Mbyte. The XL8 disk device provides up to 4.2 gigabytes of storage in a single cabinet—as many as eight drives at 520 Mbytes each.

All this capacity is packed into a footprint of only six square feet, providing storage of 420 Mbytes per square foot (including service clearance). This is a real advantage when floor space is limited.

Performance is not sacrificed to capacity. With eight actuators providing eight concurrent disk accesses, the XL8 yields very high throughput; it has an average seek time of less than 15 ms. The XL8 transfers data at 1.86 Mbytes per second, making it suitable for retrieving and storing large, sequential files in batch operations. This makes the XL8 an excellent choice for batch as well as on-line transaction processing applications.

See the accompanying articles, "Plated Media Technology Used in the XL8 Storage Facility" and "Data Encoding Technology Used in the XL8 Storage Facility," for more information.

Corinne Robinson is the product manager for Tandem's Languages and Tools. She joined Tandem in June 1983 as a software designer. Before joining Tandem, Corinne spent seven years working in microprogramming, diagnostics, and languages for another computer vendor. Corinne has a B.S. in Information and Computer Science from the University of California at Irvine.

Technical Paper: State-of-the-art C Compiler Testing

In November 1985, Tandem released its C compiler. The goal of the software development team was to produce a high-quality, reliable compiler in the shortest possible time at a reasonable cost. The application of state-of-the-art testing methods and tools played a major role in achieving this goal.

This article describes how currently available, off-the-shelf software testing tools offer a practical, cost-effective approach to thoroughly testing a C compiler. While the article is restricted specifically to Tandem's experiences in testing its C compiler, readers should find it an interesting and valuable example of what can be achieved by the use of these techniques and tools, some of which are applicable to a broader class of problems.

Increase in Available C Testing Tools

In general, during the initial phase of the testing life cycle, if a product to be tested is in widespread use or is standardized, it is worthwhile to examine current testing methods and available testing products. C meets both these criteria, as it is now in widespread use and is in the process of being standardized by the American National Standards Institute (ANSI).

As the popularity of C increases, more compiler vendors are entering the market, and the number of C programs and C programmers continues to grow. Also, a small but growing number of companies are coming forward to offer help in the design and testing of C compilers. It is no longer necessary to create a C test library entirely in-house or to rely on the outdated practice of compiling the compiler as a substitute for software quality assurance.

Common Approaches to Testing

Compiling the Compiler

An informal survey of C implementors at more than a dozen companies revealed that the most common approach to compiler testing is to compile the compiler, assuming, of course, that the compiler is written in C. This comment was often heard: "If the compiler can compile itself without producing error messages, it's time to ship." Regardless of the language in which the compiler is written, there are many reasons why this is not a sufficient approach to compiler testing.

Assume for a moment that compiling the compiler is in fact a thorough approach to C compiler testing. One would then expect the compiler source to use all of the C language features. Commonly, however, fundamental features of the language are avoided when a compiler is created. Two examples are floating point operations and bit fields. Features not used in the compiler remain completely untested when this approach to validation is used. Experience at Tandem suggests that a strong relationship exists between error-prone compiler features and the absence of those features in the compiler source itself.

Also missing from the compiler source are invalid C programs. These deviant programs, a necessary part of a thorough test library, ensure that the compiler takes the correct action when given invalid and unexpected input. Coverage statistics (presented later) indicate that without deviance tests, 15% to 25% of the compiler code is not executed.

As Myers clearly stated in 1979, one necessary component of a test case is the ability to compare the actual result to the expected result. When this comparison fails, a potential incident has been detected and is logged. Since compilers, and application programs in general, do not rigorously compare the actual result to the expected result, it is possible that a large class of errors could go undetected, even though the compiler uses a feature.

Finally, and perhaps most interesting, test coverage analysis proved that in the Tandem environment, using the compiler as a test case forced execution of only 60% of the segments in the compiler. Forty percent of the compiler was virtually unexecuted, and, thus, untested.

Compiling Applications

Many vendors who implement and sell C compilers also sell a variety of C utilities. The second tier of testing often consists of running available in-house C applications through the newly debugged compiler. Since the compiler itself is an application, all of the arguments given above for compiling the compiler hold for applications in general.

Unlike a well-written test case that logs explicit information about any difference between expected and actual results, an application may abort at compilation or execution time when encountering a compiler error. An application that leaves behind few, if any, clues about the error can be the cause of a potentially long, tedious, and costly error-isolation session.

When a well-written test case logs a potential incident, chances are good (over 90%) that the compiler is in error and that the exact nature of the error will not take long to isolate. Experience has shown that application miscues discover compiler errors a significantly lower percentage of the time.

While compiling the compiler and compiling applications do have a place in testing a compiler, companies that rely exclusively on these techniques as a substitute for software quality assurance are apt to experience a long beta test cycle and are likely to produce an unreliable compiler.

Third-party Tests

In general, testing an original software product that is under development requires a significant, original, in-house effort to create a regression test library from scratch. For C, this was the situation in the early 1970s when Dennis Ritchie designed the C programming language to aid in the development of the UNIX operating systems and their utilities (Rosler, 1984). As a result, AT&T created the first C test library.

Today, however, a vendor entering the C marketplace can expect much help in the test phases of C compiler development. Several standards of the IEEE Computer Society are now available to guide the preparation and content of documents related to testing (see IEEE standards 829-1983 and 730-1984). These worthwhile documents are useful for checking the completeness of the testing process.

C has not had the benefit of an official, formal compiler-validation facility as is available for other programming languages such as Ada, BASIC, COBOL, FORTRAN, and Pascal (Wichmann and Ciechanowicz, 1983). On the other hand, because of this lack of an official testing source, several independent companies have been formed to fill the gap, each offering a unique approach to validation.

By contacting key national testing and software quality-assurance organizations, C authors, educators, editors, implementors, ANSI representatives, utility vendors, consultants, user groups, publishers, and validation centers, the C compiler development team at Tandem discovered several generic C test suites and specific tools to aid in the creation of C tests.

Determining Test Effectiveness

After acquiring four commercially available test suites, the developers needed a practical, objective method of determining their completeness and their individual and collective contribution to the complete testing process. Practical testing methods include realistic procedures for determining when testing has been completed (Howden, 1985). Applied to compilers, test completion criteria specify when the process of executing the compiler with the intent of finding errors is judged to be complete. The most common, and yet inadequate, criteria observed in practice are (Myers, 1979):

1. Stop when all available tests fail to produce new errors.
2. Stop when the distribution-to-customers milestone arrives.
3. Stop because there is another product that should be tested immediately (or sooner).

All of these criteria are useless since they are independent of test quality; i.e., all three goals can be reached by doing absolutely nothing. A better criterion is to stop testing when over 95% of the C compiler segments have been exercised. Although 95% segment coverage might be considerably more difficult to achieve for an Ada compiler, this goal is realistic for most C compilers, considering their size.

Test Coverage Analysis

The Test Coverage Analysis Tool (TCAT) for C aids in investigating the effectiveness of program testing.[1] TCAT expresses test coverage in terms of segments exercised and not exercised. A segment is a basic block of consecutive statements that may be entered only at the beginning and that, when entered, are executed in sequence without halt or possibility of branch (except at the end of the basic block).

Every executable statement is in a segment that corresponds to an edge in the program's directed graph. Each segment has only one entry and one exit node. TCAT measures the extent to which one test or a test suite exercises all of the segments in a program (i.e., a C compiler).

TCAT Results

The Tandem C software development team used TCAT to determine the effectiveness of the four test suites. Table 1 summarizes the percentage of segments each suite exercised. Suite A consists of the programs in *The C Puzzle Book* (Feuer, 1982). Programmers in various computer companies created the other suites by going through Appendix A of Kernigham and Ritchie's *The C Programming Language* and hand-coding tests.

The Cumulative Coverage column includes the contribution from the entry on a given line plus the contribution from each previous line. For example, suites A, B, and C combined yield a coverage of 70%. All four suites combined yield a coverage of 76%.

It is interesting that although all four suites were created independently, their overlap with respect to segment coverage is considerable. In fact, Suite D, when combined with the three other suites gains only 2% more segments compared to its coverage alone.

Table 1.
Segment coverage of four C compiler test suites.

Test suite	Segments hit	Stand-alone coverage	Cumulative coverage
Suite A	1567	57%	57%
Suite B	1754	64%	66%
Suite C	1863	68%	70%
Suite D	2023	74%	76%

[1]TCAT was developed by Edward Miller, Software Research Associates, P.O. Box 2432, San Francisco, CA 94126.

These numbers say nothing about unique paths through the code or unique sets of input data. There is evidence, however, that the suites are more different than simple segment-coverage measurements indicate. Early in the development cycle, when many fundamental errors were present, several or all of the test suites would often discover the same error. Later in the cycle, as the product began to mature, it became more frequent for only one of the test suites to discover new errors.

One reason for this is that each suite tended to closely follow a particular coding style that was consistent throughout the suite, but which varied from suite to suite. While there are advantages to having a test library composed of functionally overlapping test cases written by different people, this is often not possible for economic reasons. Testers would do well to employ as much randomization as possible, however (i.e., try to test the code in as many ways as possible). This also supports the idea of product developers performing their own unit tests while independent testers create the test suites in parallel. Having either a product developer or testing developer do all the testing is insufficient.

As is clear in Table 1, although Suite D leaves 26% of the compiler untested, it is superior to the other test suites that were also designed without the guidance of a coverage tool. Typical programmers who do not have the benefit of detailed coverage analysis normally produce test suites that cover only 25% to 50% of the segments (see Miller, 1984). Thus, all the programmers who created the above suites must be above average. One reason for the higher coverage obtained by Suite D is its developer's understanding of the importance of deviance test cases, an ingredient missing from the others.[2]

Since all four suites combined still left 24% of the compiler untested, it was clear that the test completion goal of 95% was not satisfied. Details on what was needed to increase the coverage to 95% are included in the next section.

Missing Tests

As Table 1 indicates, independent programmers, without the aid of a coverage tool and specific test completion criteria, decided when the testing tasks in these suites was complete. Using the specific feedback from the TCAT coverage analysis, one Tandem developer needed only one month to increase the test coverage to exceed 95%.

The following are the less obvious kinds of test case that are easy to detect with the use of a coverage tool but easy to miss without one.

Binary Expressions with Constant Operands. For example, a good mix of short, unsigned, long, and double operands combined with a variety of the operators *, +, >>, <<, <, >, <=, >=, ==, !=, &, ^, and | is useful. In this context, operands are constant.

Bit-field Tests. In particular, operations on bit fields in arithmetic expressions should be tested. For example, field tests should ensure that the -, ~, !, *, /, %, >, <, ==, &, ^, |, &&, ||, ?, ++, and -- operators work correctly in expressions containing bit fields.

Combination Tests. A suite that does an excellent job of testing individual features, but lacks many more tests that combine the features, is insufficient. For example, an astronomical number of expressions are possible that contain up to 32 random operators using random data types for operands. Ideally, separate, machine-generated programs should be created for the combination tests. Fortunately, a self-checking C expression generator is commercially available.[3]

[2]Perennial Software Services Group provides the C Compiler Validation Suite, represented in this article as Suite D. Their address is 3130 De La Cruz Blvd., Santa Clara, CA 95054.

[3]The self-checking C expression generator is available from Ralph A. Phraner and Associates, 516 Shrader Street, San Francisco, CA 94117.

Preprocessor Features. This aspect of the language is tempting to neglect during testing. The preprocessor is a critical part of the language, however, and must be tested. Implementing the preprocessor is a difficult programming task that consumes a significant part of the source code comprising any good C compiler. Developers must be sure to include tests for *#else* that involve nested *#if*, *#ifdef*, and *#ifndef*, and tests for *#undef*, as well as error conditions within preprocessor commands.

Deviance Tests. As mentioned earlier, these are programs that differ from standard C in some way, for example:

- A *do* statement missing the *while* clause.
- A *goto* statement missing the label from the *goto*.
- A *goto* statement having a label whose name is the same as a local variable.
- A *#define* preprocessor command containing a premature end of file.

Library Tests. Early specifications of the C language did not incorporate the run-time library routines. Since ANSI has incorporated the library into the language, a C test suite that ignores the library functions specified by ANSI is severely deficient.

Conversions. Although *int* is typically well covered, tests are needed that contain expressions using operands of different types within the same expression (e.g., an expression mixing operands of type *long*, *float*, and *double*). In addition, tests that force conversions involving pointers are useful.

Compiler-option Tests. This area is also tempting to neglect, but compiler users detest easy discovery of options that do not perform as documented.

for *Statement.* A *for* statement having a test value (a second expression) that is zero is useful.

***Keyword* default.** The test library should include a test case that tests for proper operation of the optional keyword *default*.

Bit-field Initialization. Initializing static structures containing bit fields is useful.

Expression. An expression that contains a function call using call-by-reference parameters is valuable as a test component.

#if *Preprocessor Command.* Also useful is an *#if* preprocessor command that contains a constant expression using a hexadecimal constant, an octal constant, a character constant, ~ (a tilde, the one's complement operator), %, /, <, >, !, =, &, ^, |, :, ?, (, and). (These do not necessarily need to be used in the same command.)

Hexadecimal Constants. A variety of escape sequences that contain hexadecimal constants (the hex code following a backslash), in which the hex constants contain a mixture of digits, uppercase *A* through *F*, and lowercase *a* through *f*.

Limits Tests. The ANSI C Draft Standard specifies many minimal limits that must be met or exceeded and, therefore, should be tested; e.g., *#include* is limited to nesting levels of eight or more.

Testing Compiler Performance

In addition to testing the features of their C compiler, vendors must ensure that the compiler produces fast object code and that it compiles quickly. The importance of providing an ability to quantitatively assess the compiler's performance before each release and compare it with previous releases may not be as obvious. Programs such as the Sieve of Eratosthenes and Fibonacci number generation, as well as other benchmark test cases covering a variety of language constructs, are readily available from the literature (Leibson, et al., 1985). These programs are easily added as performance test cases.

Shorter Alpha- and Beta-test Phases

A common misconception among software producers is that complete testing means more cost to the vendor and a longer development cycle. On the contrary, the use of a thorough internal test library substantially reduces the length of the alpha- and beta-test phases without lengthening any other development phase. A fundamental function served by the beta test is to confirm that the product is well designed and tested; if it has been, a six-week beta test should suffice to confirm its quality.

A poorly tested compiler requires beta-test users to discover errors. This results in several rounds of testing, each of which introduces a new version of the compiler to correct errors found in the previous one. Each round of the beta test requires the time to release the latest version; distribute it to the users, wait for them to install it, find errors, and communicate them; attempt to decipher the often cryptic and perhaps erroneous information; and correct the real errors. It should be obvious that most of this wasted time would be avoided by conducting several considerably shorter rounds of in-house testing before beginning the alpha test.

Since, for one compiler update, one round of testing can easily take three months during the beta-test phase, a compiler that is beta-tested without the benefit of a good in-house test library could spend a long time in that phase. While one round is often sufficient for a solid product, three to six (and perhaps more) should be expected for a compiler that has been tested minimally in-house. Thus, if four rounds of testing are needed, a product could spend over a year in the beta-test phase, resulting in a loss of revenue and customers.

The Tandem C Compiler was released after less than six weeks in the first and only round of beta testing. No serious release-stopping software errors were discovered in the beta test or in the several months that followed the first customer shipment.

Conclusion

Applying currently available testing tools shortens the development time of a C compiler and increases the quality of the product. It is essential that the testing process occur in parallel with the development process.

C compiler vendors can avoid the losses resulting from inadequate testing by employing the skills of a permanent, well-trained software quality-assurance staff and a complete and appropriate library of compiler testing tools. Traditional testing approaches, such as compiling the compiler, are less than satisfactory and can now be replaced with reliable C testing tools.

References

ANSI/IEEE Standard for Software Quality Assurance Plans. 1984. IEEE std. 730-1984. IEEE Press. (Revision of ANSI/IEEE std. 730-1981.)

ANSI/IEEE Standard for Software Test Documentation. 1983. IEEE std. 829-1983. IEEE Press.

Feuer, A.R. 1982. *The C Puzzle Book*. Prentice-Hall, Inc.

Howden, W.E. 1985. The Theory and Practice of Functional Testing. *IEEE Software*. Vol. 2, No. 5.

Kernigham, B., and Ritchie, D. 1978. *The C Programming Language*. Prentice-Hall, Inc.

Leibson, S., Pfahler, F., Reed, J., and Kyle, J. 1985. Software Reviews: Expert team analyzes 21 C compilers. *Computer Language*. Vol. 2, No. 2.

Miller, E., et al. 1984. *User's Manual for TCAT/C (PC Version)*. Software Research Associates.

Myers, G.J. 1979. *The Art of Software Testing*. John Wiley and Sons.

Rosler, L. 1984. The Evolution of C—Past and Future. *AT&T Bell Laboratories Technical Journal*. Vol. 63, No. 8.

Wichmann, B.A., and Ciechanowicz, Z.J. 1983. *Pascal Compiler Validation*. John Wiley and Sons.

Ed Kit is a member of Software Quality Assurance within the Languages Group of Software Development. Since joining Tandem in 1980, his responsibilities have included creating test suites for communication protocols, languages, and terminals, and managing performance, data-base, operating systems, data communication, and microcode software quality-assurance groups. Previously he was on the faculty of Embry-Riddle Aeronautical University, where he taught computer science, mathematics, and electrical engineering. Ed holds a B.S. and M.S. in Electrical Engineering from Purdue University.

TANDEM PUBLICATIONS ORDER FORM

The *Tandem Systems Review* and the Tandem Application Monograph Series are combined in one free subscription. Use this form to subscribe, change a subscription, and order back copies.

For requests *within the U.S.*, send this form to:

Tandem Computers Incorporated
Sales Administration
19191 Vallco Parkway, MS 4-05
Cupertino, CA 95014-2599

For requests *outside the U.S.*, send this form to your local Tandem sales office.

Check the appropriate box(es):

- ☐ New subscription (# of copies desired _____)
- ☐ Subscription change (# of copies desired _____)
- ☐ Request for back copies. (Shipment subject to availability.)

Print your current address here:

_____
ADDRESS

_____

_____

_____

_____

_____
ATTENTION

_____
PHONE NUMBER (U.S.)

If your address has changed, print the old one here:

_____
ADDRESS

_____

_____

_____

_____

_____
ATTENTION

_____
PHONE NUMBER (U.S.)

To order back copies, write the number of copies next to the title(s) below.

NUMBER OF COPIES

Tandem Journal

_____ Part No. 83930, Vol. 1, No. 1, Fall 1983
_____ Part No. 83931, Vol. 2, No. 1, Winter 1984
_____ Part No. 83932, Vol. 2, No. 2, Spring 1984
_____ Part No. 83933, Vol. 2, No. 3, Summer 1984

Tandem Systems Review

_____ Part No. 83934, Vol. 1, No. 1, February 1985
_____ Part No. 83935, Vol. 1, No. 2, June 1985
_____ Part No. 83936, Vol. 2, No. 1, February 1986
_____ Part No. 83937, Vol. 2, No. 2, June 1986

Tandem Application Monograph Series

_____ Part No. 83900, *Developing TMF-Protected Application Software,* March 1983, AM-005
_____ Part No. 83901, *Designing a Tandem/Word Processor Interface,* March 1983, AM-006
_____ Part No. 83902, *Integrating Corporate Information Systems: The Intelligent-Network Strategy,* March 1983, AM-007
_____ Part No. 83903, *Application Data Base Design in a Tandem Environment,* August 1983
_____ Part No. 83904, *Capacity Planning for Tandem Computer Systems,* October 1984
_____ Part No. 83905, *Sociable Systems: A Look at the Tandem Corporate Network,* May 1985
_____ Part No. 83906, *Transaction Processing on the Tandem NonStop Computer: Requestor/Server Structures,* January 1982, SEDS-001
_____ Part No. 83907, *Designing a Network-Based Transaction-Processing System,* April 1982, SEDS-002

TANDEM EMPLOYEES: PLEASE ORDER YOUR COPIES THROUGH YOUR MARKETING LITERATURE COORDINATOR.

TANDEMCOMPUTERS

Part No. 83937

400099 06/86 Printed in USA