# TANDEMJOURNAL

*A New Design for the PATHWAY TCP*

*Understanding PATHWAY Statistics*

*A SNAX Passthrough Tutorial*

*The TRANSFER Delivery System For Distributed Applications*

# T A N D E M J O U R N A L

**Volume 2, Number 2, Spring 1984**

**Editor**
Carolyn Turnbull White

**Associate Editor**
Kent Madsen

**Production Editor**
Anita Van Auken

**Technical Advisor**
Geary Arceneaux

**Design**
Craig Frazier Design

**Cover Art**
Craig Frazier

The Tandem Journal is published quarterly by Tandem Computers Incorporated.

**Purpose:** The purpose of the Tandem Journal is to bring to Tandem users the perspectives of Tandem software developers, engineers, and support analysts on Tandem software and hardware.

**Subscriptions:** The Tandem Journal is offered with the Tandem Application Monograph Series in one subscription. The annual subscription rate is $100.00. Tandem bills the subscriber. *U.S. Orders* — Send directly to Tandem Computers Incorporated, Sales Administration, 19333 Vallco Parkway, Cupertino, CA 95014. *Orders outside the U.S.* — Give to your local Tandem sales office or distributor. All subscribers should address subscription problems or questions to their local Tandem sales office or distributor.

**Change of address:** Send all changes of address to Sales Administration (address listed above).

**Comments:** We welcome comments and suggestions about content and format. Please send them to Carolyn Turnbull White, Editor, Tandem Journal, Tandem Computers Incorporated, 1309 So. Mary Ave., Sunnyvale, CA 94087.

# A New Design for the PATHWAY TCP

In terms of system resource use, disc I/O operations are the most expensive Terminal Control Process (TCP) operations that the PATHWAY™ transaction processing system performs. Therefore, to improve performance, the PATHWAY system development team has chosen to concentrate its initial effort on reducing the number of TCP disc I/O operations.

Until the E07 release of the PATHWAY system, TCP disc I/O operations had been reduced in two ways: by increasing the maximum size of a disc I/O operation and by allowing the TCP to bypass searching the directory file while executing a Screen COBOL CALL command, if the called program unit was already in memory.

In the E07 release, Tandem has provided a further and more significant TCP improvement: the use of an extended segment to replace the disc swap file. As a result, two versions of the TCP are now available (referred to in this article as TCP1 and TCP2):

- TCP1 is essentially the same TCP released in previous versions of the PATHWAY system. It runs on the NonStop™, NonStop II™, and NonStop TXP™ systems.
- TCP2 uses an extended segment and runs on the NonStop II and NonStop TXP systems.

This article describes the main differences in implementation between the two TCPs. It explains why TCP1 executes disc I/O operations and how TCP2 eliminates most of them, primarily by replacing the disc swap file with the extended segment. It also describes the performance characteristics of TCP2, and the performance improvement of TCP2 over TCP1. Finally, it explains why TCP2 addresses the most important factor in the performance of the PATHWAY system.

The implementation described in this article is logical in nature. For conciseness, many details are omitted, and some precision is compromised.

## Memory Organization

The TCP is a multi-tasking process running under the GUARDIAN™ operating system. TCP1 has only 128K bytes of data address space to work with. Figure 1 illustrates how this address space is organized.

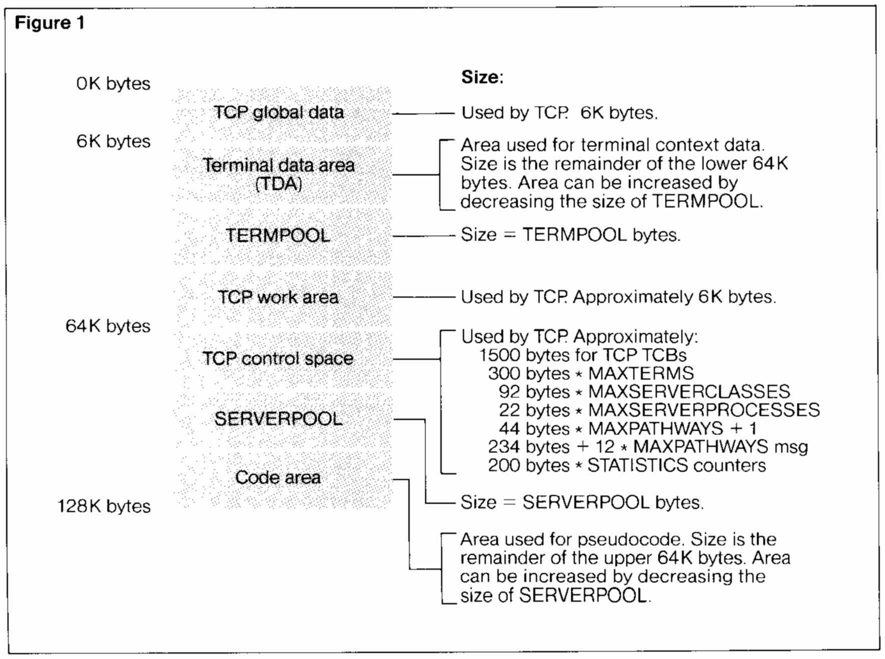


Figure 1
0K bytes
6K bytes
64K bytes
128K bytes
TCP global data
Terminal data area (TDA)
TERMPOOL
TCP work area
TCP control space
SERVERPOOL
Code area
Size:
Used by TCP. 6K bytes.
Area used for terminal context data. Size is the remainder of the lower 64K bytes. Area can be increased by decreasing the size of TERMPOOL.
Size = TERMPOOL bytes.
Used by TCP. Approximately 6K bytes.
Used by TCP. Approximately:
1500 bytes for TCP TCBs
300 bytes * MAXTERMS
92 bytes * MAXSERVERCLASSES
22 bytes * MAXSERVERPROCESSES
44 bytes * MAXPATHWAYS + 1
234 bytes + 12 * MAXPATHWAYS msg
200 bytes * STATISTICS counters
Size = SERVERPOOL bytes.
Area used for pseudocode. Size is the remainder of the upper 64K bytes. Area can be increased by decreasing the size of SERVERPOOL.

**Figure 1.**
*Memory organization for TCP1.*

TCP2 uses an extended segment in addition to the 128K of address space in its stack. The organization of address space for TCP2 is illustrated in Figure 2.

As is clear from a comparison of Figures 1 and 2, TCP2 has a much larger address space than TCP1 and more space for TERMPOOL and SERVERPOOL. This decreases the likelihood that a terminal task will be blocked from execution due to a shortage of pool space.

## TCP1 Disc I/O Operations

TCP1 executes disc I/O operations to perform:

- Context swapping.
- Checkpointing.
- Code fetching.

Figure 3 illustrates these three activities.

The swap file assigns two slots of address space for each terminal task. Each slot is the size of MAXTERMDATA + MAXREPLY. Although the actual implementation is not this simple, logically, SLOT0 can be thought of as holding the current context data for the task when the context is not in memory. SLOT1 can be thought of as holding the most recent checkpoint image of the task.

### Context Swapping

Context swapping is the movement of context data for a terminal task between the swap file and the Terminal Data Area (TDA) for TCP1. Swapping serves no useful application function. It is necessary if the TDA is smaller than the sum of the context data of all the active tasks.

For example, in Figure 3, if Task B is about to execute, its context data must be present in the TDA. If the context data is not present, it is fetched from slot B0 of the swap file. The context for Task B stays in the TDA until the space it occupies is needed for the context of another task that is about to execute. When the space holding the context for Task B is needed by another task, TCP1 moves all of Task B's context from the TDA back to slot B0.

Figure 2

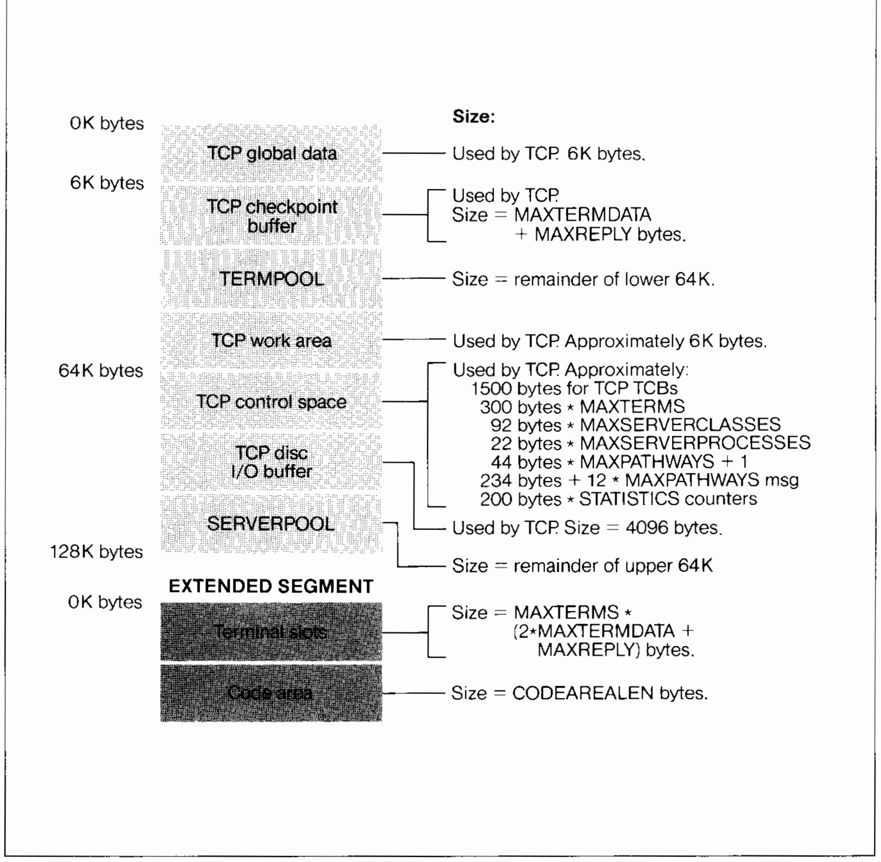

0K bytes
6K bytes
64K bytes
128K bytes
0K bytes
Size:
TCP global data
Used by TCP. 6K bytes.
TCP checkpoint buffer
Used by TCP. Size = MAXTERMDATA + MAXREPLY bytes.
TERMPOOL
Size = remainder of lower 64K.
TCP work area
Used by TCP. Approximately 6K bytes.
TCP control space
Used by TCP. Approximately: 1500 bytes for TCP TCBs 300 bytes * MAXTERMS 92 bytes * MAXSERVERCLASSES 22 bytes * MAXSERVERPROCESSES 44 bytes * MAXPATHWAYS + 1 234 bytes + 12 * MAXPATHWAYS msg 200 bytes * STATISTICS counters
TCP disc I/O buffer
Used by TCP. Size = 4096 bytes.
SERVERPOOL
Size = remainder of upper 64K
EXTENDED SEGMENT
Terminal slots
Size = MAXTERMS * (2*MAXTERMDATA + MAXREPLY) bytes.
Code area
Size = CODEAREALEN bytes.

Figure 3

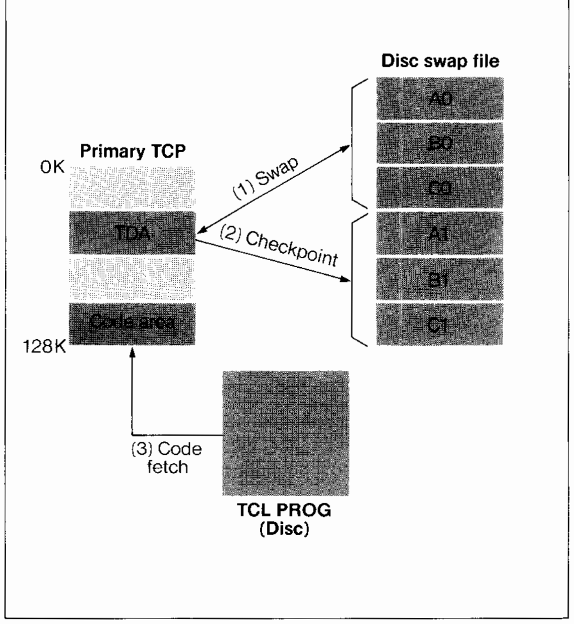


**Figure 2.**
*Memory organization for TCP2.*

**Figure 3.**
*The three types of disc activity for TCP1: (1) swapping the task context data between the disc swap file and the Task Data Area, (2) writing the task context data or server reply from the Task Data Area to the disc swap file during a checkpoint operation, and (3) reading Screen COBOL code from the program file on disc to the code area in memory. (In this example, MAXTERMS = 3.)*

**Checkpointing**
Checkpointing is the saving of context data or the server reply for the currently executing task. It is done for these reasons:

▪ If the primary TCP fails, the backup TCP takes over for each task by using the context image found in SLOT1 of the task.

▪ If a transaction protected by the Transaction Monitoring Facility (TMF) fails, or if a RESTART-TRANSACTION is executed, the primary TCP restores the state of the task using the BEGINTRANSACTION checkpoint image saved in SLOT1.

TCP1 does not use the standard checkpoint mechanism provided by GUARDIAN. Instead, it saves a task's checkpoint data in SLOT1 of the task in the swap file. Several events cause TCP1 to perform a checkpoint to SLOT1. One is the execution of the BEGINTRANSACTION statement. The following also cause a checkpoint, if the TCP is designated as a NonStop TCP (SET TCP NONSTOP 1):

▪ Execution of the CHECKPOINT statement.

▪ Execution of the ENDTRANSACTION statement.

▪ Execution of the SEND statement while the TCP is communicating outside transaction mode with a server unprotected by TMF. (In this case, TCP1 saves the context before the send to the server and saves the reply after the reply arrives from the server.)

▪ Execution of the RECONNECT MODEM statement.

Figure 4

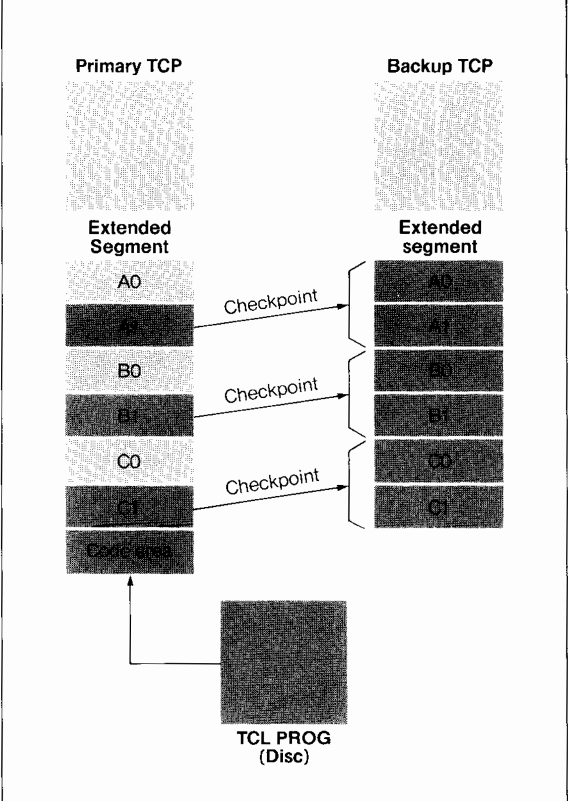


**Figure 4.**
*With TCP2 no swapping takes place, and checkpointing requires one interprocess message instead of one or more disc I/O operations. While code fetching still takes place, the code area for TCP2 can be much larger than that for TCP1. (In this example, MAXTERMS = 3.)*

**Code Fetching**
Code fetching is the reading of Screen COBOL pseudocode or screen descriptions by TCP1 from the object file into the code area.

Whenever possible, TCP1 uses the free space in its code area for Screen COBOL pseudocode and screen descriptions. If no free space is available, some segments in the code area must be overlaid with the incoming code segment. If TCP1 needs the overlaid segments later and has to read them in again, overlaying the code segment is a wasteful operation.

## TCP2 Disc I/O Operations

TCP1 requires a disc swap file because its address space is limited to 128K bytes. The context data for all user tasks cannot necessarily reside in the TCP1 address space concurrently. Therefore, the private address space of all the terminal tasks is kept in the swap file (in SLOT0s).

TCP2 does not need a swap file because it has an arbitrarily large address space in the form of an extended segment. Users can instruct TCP2 to allocate an extended segment large enough to hold the context data for all terminal tasks concurrently. Figure 4 illustrates TCP2 memory organization for the same configuration shown in Figure 3.

Each TCP2 process assigns two slots of address space for each terminal task. SLOT0 is the size of MAXTERMDATA. SLOT1 is the size of MAXTERMDATA + MAXREPLY. The functions of SLOT0 and SLOT1 in the primary TCP2 are analogous to their SLOT0 and SLOT1 counterparts in the swap file used by TCP1. In TCP2, SLOT0 contains the current context data for the task. SLOT1 contains the last checkpoint image. The TDA does not exist in TCP2 (see Figure 2).

**Context Swapping**
TCP2 eliminates the disc I/O operations performed by TCP1 for context swapping and checkpointing. TCP2 does no context swapping because it acts directly on the task's context in SLOT0. TCP2 checkpointing requires one interprocess message instead of one or more disc I/O operations.

**Checkpointing**
In a checkpoint operation, TCP2 saves the checkpoint image in SLOT1 of the primary TCP. If the TCP is designated as a NonStop TCP, it sends the checkpoint image to one of the task's SLOTs in the backup TCP. The ENDTRANSACTION checkpoint causes TCP2 to send the checkpoint image to the task's SLOT0 in the backup TCP; all other checkpoints cause it to send the images to the task's SLOT1 of the backup TCP.

The reasons for having SLOT1 in the primary TCP2 are:

- A BEGINTRANSACTION image must always be saved, in case the transaction must be restarted.
- If the backup TCP2 dies and is recreated later, the primary TCP2 must be able to send the checkpoint images of all the active tasks to the backup TCP2 immediately. (Otherwise, the primary TCP2 would have to wait for each task to reach its next checkpoint state.)

The ENDTRANSACTION checkpoint image is stored in backup SLOT0 while the BEGINTRANSACTION checkpoint image is stored in backup SLOT1. If the primary TCP2 fails at a point after the completion of the ENDTRANSACTION checkpoint, but before the call to ENDTRANSACTION, the backup must take over at the BEGINTRANSACTION image. (Tony Lemberger's article, "TMF and the Multi-Threaded Requester," published in the Fall 1983 issue of the *Tandem Journal* explains this in more detail.)

**Code Fetching**
TCP2 still performs disc I/O operations for code fetching. However, users can specify an arbitrarily large code area in the TCP2 extended segment, and if the code area is large enough, TCP2 need read each code segment from disc only once.

## Performance Implications

The development group conducted extensive performance measurements on the TCP in order to understand its behavior and to determine the best ways to improve the performance of the PATHWAY transaction processing system. After TCP2 was implemented, both this group and Tandem's Product Management Technical Services group independently measured the performance differences between TCP1 and TCP2. While it is beyond the scope of this article to present their measurement methods and the resulting statistics, it is possible to present some conclusions.

**TCP2 versus TCP1**
TCP2 is faster than TCP1. It performs better because the number of disc I/O operations it executes has been reduced.

The most important yardsticks for measuring performance are throughput and response time. These parameters were measured in terms of transactions protected by TMF. For any given throughput, TCP2 has a lower transaction response time because it executes fewer disc I/O operations per transaction.

It is important to understand that a disc I/O initiated by the TCP consumes not only TCP resources but resources throughout the system. This makes it impossible to compare the performance of the two TCPs as stand-alone entities, and incorrect to compare only the TCP statistics generated by the XRAY performance measurement tool.

*TCP2 performs better than TCP1 at less cost.*

The development groups compared TCP2 against TCP1 by running the same application under environments that were identical except for the TCPs. While keeping transaction arrival rates constant, they compared the transaction response times under the two TCPs. In all tests, response time for TCP2 was lower than for TCP1.

Of course, the amount of performance improvement for TCP2 depends on many factors. System configuration and application characteristics are two important examples.

**Context Size**

The amount of improvement for TCP2 depends greatly on the context size of the Screen COBOL application. The greater the context size, the more efficient is TCP2 than TCP1. Context size affects the performance of TCP1 in two ways:

1. A large context increases the likelihood of swapping.
2. A large context increases the number of disc I/O operations required for swapping and checkpointing operations.

A disc I/O operation can move 4K bytes at most. For each transaction protected by TMF, a TCP1 task with 10K bytes of context requires three disc I/O operations for checkpointing at BEGINTRANSACTION and three disc I/O operations for checkpointing at ENDTRANSACTION. A similar TCP2 task requires two interprocess messages to achieve the same result. Regardless of context size, TCP2 sends a single interprocess message for each checkpoint operation. Thus, TCP2 performance is, for the most part, unaffected by context size.

**Interpreter Performance**

The amount of improvement for TCP2 depends on the number of Screen COBOL instructions executed in a transaction. The TCP2 interpreter is slightly slower than its counterpart in TCP1 because it acts on SLOT0 in the extended segment. For various system reasons, some TAL instructions affecting extended segments are slightly slower than those affecting the stack. (Some of these system reasons do not apply to the NonStop TXP system.)

This means that a transaction that executes many Screen COBOL instructions realizes less improvement under TCP2 than a transaction that executes fewer Screen COBOL instructions, if these transactions are otherwise the same.

**The Cost of a NonStop TCP**

To improve the performance of the TCP, some TCP1 users choose not to have a NonStop TCP. This is a false economy with TCP2 because the cost of having a NonStop TCP2 is basically a single interprocess message for each checkpoint operation.

**Memory Cost Versus Disc Cost**

The major cost associated with TCP2 is the extended segment. Using a large extended segment increases the likelihood of page faulting by the GUARDIAN operating system. When TCP2 requests an extended segment, GUARDIAN must allocate a disc file the same size as the segment. This file is used by GUARDIAN in case paging occurs for pages in the extended segment. Disc space must be available for this purpose even if no disc activity to this file is necessary.

Users must avoid page faults to realize the full benefit of TCP2. Careful system tuning is required to monitor paging. In many instances, more physical memory is required. The savings in disc activity resulting from not having a swap file normally offsets this memory cost, as was verified by two independent measurements of different applications.

### Disc I/O Cost Versus Interpretation Cost

By minimizing the number of disc I/O operations, TCP2 eliminates the most costly factor in the performance of the PATHWAY system. A disc I/O operation contributes significantly to response time because it involves: processing in the TCP to initiate the I/O, an interprocess message to the Discprocess, processing in the Discprocess to perform the I/O, service time on the part of the disc device while it performs mechanical operations, processing to handle I/O terminations, and various queuing delays.

There is a common misconception that interpretation of Screen COBOL by the TCP is the most costly performance factor. This is generally not true because the overhead of interpretation is extra processing in the TCP process only. For any given throughput, this extra processing generally does not contribute as much to response time as do disc I/O operations. There are no extra interprocess messages, no queuing delays, and no additional demands on the operating system. Furthermore, the most expensive Screen COBOL operations (screen verbs such as ACCEPT and DISPLAY, transaction control verbs that cause checkpoints, and SEND) are functions that have no counterparts in TAL or COBOL. These operations would not be much faster even if they were not executed from an interpreter.

For the reasons discussed in this section, the PATHWAY development team decided to concentrate on minimizing the number of disc I/O operations. Their design decisions and measurement methods were based on the assumption that a typical application using the PATHWAY transaction processing system incurs disc I/O operations in the TCP. They also assumed that a reasonable application runs transactions that are protected by TMF and a NonStop TCP. It is, of course, possible to construct an application that would not benefit from using TCP2, but this application would have to forgo checkpointing, cause no context swapping and code overlaying, execute many Screen COBOL instructions for each transaction, and require low throughput.

## Conclusion

TCP2 differs from TCP1 in two significant ways. It replaces the disc swap file with an extended segment and allows an arbitrarily large code area in an extended segment. It performs better than TCP1 because it executes fewer disc I/O operations per transaction. TCP2 eliminates disc I/O operations caused by checkpointing and context swapping and keeps code fetching to a minimum. Minimizing disc I/O operations eliminates the most costly performance factor for the PATHWAY system.

The performance of TCP2 is not sensitive to context size or to code size. This makes it easier to configure TCP2 and to write modular Screen COBOL programs. A NonStop TCP is inexpensive under TCP2. Transaction response time, rather than XRAY statistics for the TCP, should be used in measuring TCP performance.

TCP2 requires more memory, but less disc. Since TCP2 delivers better performance at less cost, it is clear that users of the NonStop II and NonStop TXP systems should migrate to TCP2 if they would like better performance from their applications. Future performance enhancements to the PATHWAY transaction processing system will continue the trend of using more memory to obtain better performance.

**Raymond Wong** joined Tandem in March, 1981. Since then he has been a software developer for the PATHWAY system, primarily for the TCP. He has worked in system software for ten years, in the areas of communications, operating systems, and languages.

# Understanding PATHWAY Statistics

The statistics produced by the PATHWAY transaction processing system represent various aspects of resource usage by the Terminal Control Process (TCP). This information can be used to detect some performance bottlenecks and configuration errors. This article describes the general implementation of the TCP and explains the statistics. It does not discuss how to use the statistics to improve the performance of a particular application.

The statistics explained are for TCP2, the new version of the PATHWAY TCP available in the E07 release.[1] The description of TCP2 provided in the previous article, "A New Design for the PATHWAY TCP," should also be helpful in understanding TCP statistics.

PATHWAY statistics are divided into three categories: TCP statistics, terminal statistics, and server statistics. The first section discusses TCP statistics.

## TCP Statistics

### System Tasks

The TCP is a multi-tasking process running under the GUARDIAN operating system. The number of terminal tasks supported by a TCP is specified by the MAXTERMS parameter of the SET TCP command. In addition to the terminal tasks, every TCP has four system tasks: the Checkpointer, the Code Manager, the Listener, and the Speaker. Refer to Figure 1 as these tasks, the queues associated with them, and the pool areas are discussed below.

***The Checkpointer.*** A task must send information to its counterpart in the backup TCP so that the backup can take over if the primary TCP fails. It requests the Checkpointer to provide this service. The information sent may be the task context, server replies, the Screen COBOL object file name, or other items. For example, after a terminal task opens a terminal during START TERM, it sends the symbolic name of the terminal to the backup TCP so that the backup can open the same terminal immediately.

The Checkpointer serves the tasks in the TCP one at a time. A TCP task requests service by placing itself on the request queue associated with the Checkpointer. It is then blocked from execution while waiting for service and is removed from the request queue and awakened by the Checkpointer when the service completes.

Figure 2 illustrates a sample set of TCP statistics. The CHECKPOINT entries under QUEUE INFO of the TCP statistics describe the request queue. REQ CNT (request count) indicates the number of requests made to the Checkpointer. The % WAIT entry indicates the percentage of requesting tasks that find other tasks already on the request queue at the time of the request. MAX WAITS indicates the longest queue encountered by any requesting task before it joined the queue. AVG WAITS indicates the average number of tasks on the queue.

[1]The TCP statistics and terminal statistics for TCP2 are different from those for previous versions of the TCP. Server statistics are the same for all recent versions of the TCP.

Figure 1

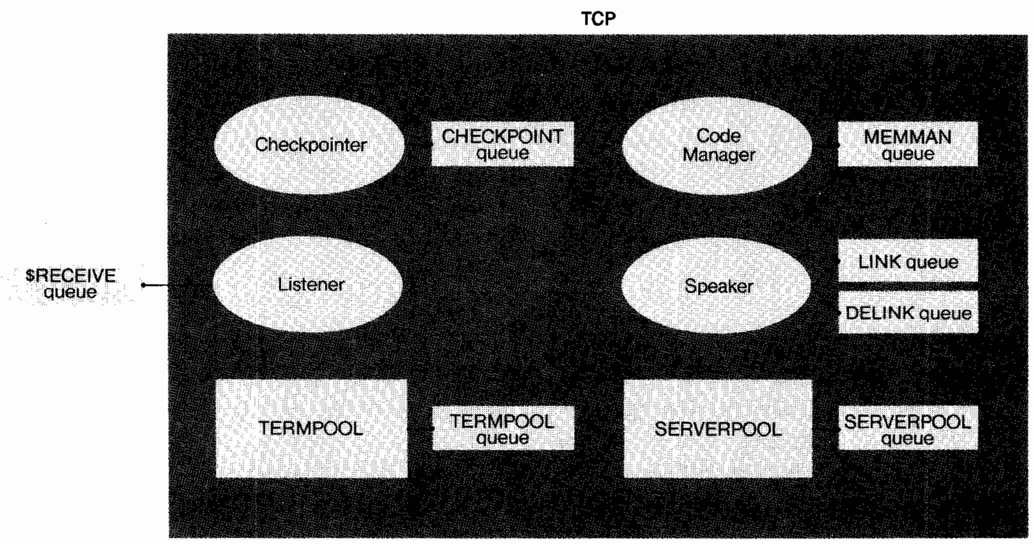


**Figure 1.**
*A graphic representation of the queues described by the QUEUE INFO portion of the Terminal Control Process statistics (part of the statistics produced by the PATHWAY transaction processing system). The queues are associated with three system tasks (the Speaker, Listener, and Checkpointer) and two pool areas (TERMPOOL and SERVERPOOL). See Figure 2 also.*

***The Code Manager.*** The Code Manager's primary function is to fetch Screen COBOL object code from disc to the TCP code area in memory. Like the Checkpointer, the Code Manager provides service to other TCP tasks and has a request queue. The MEMMAN (memory manager) entries under QUEUE INFO describe the queue and have the same meaning as those described for CHECKPOINT.

***The Listener and Speaker.*** The Listener and Speaker communicate with the PATHWAY Monitor (PATHMON) on behalf of the TCP. PATHMON and the TCP communicate over two channels. On one channel, the Listener for the TCP accepts requests from PATHMON and later replies to PATHMON. On the other channel, the Speaker for the TCP sends requests or error indications to PATHMON, which replies to the TCP.

Figure 2

```
TCP   TCP-A                                                16 DEC 1983, 10:18:29
 POOL INFO:          SIZE      REQ CNT    MAX ALLOC    AVG ALLOC    CUR ALLOC
   TERMPOOL         34818            7         1016          456           12
   SERVERPOOL       55706            1           44           44            0
                  MAX REQ      AVG REQ
   TERMPOOL          1016          456
   SERVERPOOL          44           44
 AREA INFO:          SIZE      REQ CNT    MAX ALLOC    AVG ALLOC    CUR ALLOC
   DATA            153600           --         2182           --         1254
   CODE             65536           19          848          360          844
                  MAX REQ      AVG REQ     % ABSENT
   DATA                --           --           --
   CODE               648          334         15.7
 QUEUE INFO:      REQ CNT       % WAIT    MAX WAITS    AVG WAITS
   TERMPOOL             7          0.0            0         0.00
   SERVERPOOL           1          0.0            0         0.00
   MEMMAN               3          0.0            0         0.00
   LINK                 1          0.0            0         0.00
   DELINK               1          0.0            0         0.00
   CHECKPOINT           5          0.0            0         0.00

-- = TCP2 does not generate figures for these fields.
```

**Figure 2.**
*A sample set of Terminal Control Process statistics produced by the PATHWAY transaction processing system.*

The Listener receives requests from PATHMON, initiates TCP action to serve the requests, and replies to PATHMON. The Listener is invoked by requests from PATHMON, and its request queue is the TCP's $RECEIVE. There are no statistics for the Listener's request queue because $RECEIVE is not an internal TCP queue.

The Speaker sends unsolicited information to PATHMON. It also performs most of the work of managing links to servers. Two of the most frequent requests to PATHMON from the TCP are requests to create a new link to a server class and requests to dissolve an existing link. A LINK queue and a DELINK queue are associated with the Speaker for these requests. (See the section on server statistics for a more complete description of link management by the TCP.)

*Two of the TCP's most frequent requests to PATHMON are to create a new link to a server class and to dissolve an existing link.*

Tasks on the LINK queue are terminal tasks. If a terminal task decides to ask PATHMON to create a new link, it specifies a (possibly 0) time-out value and places itself on the LINK queue. The Speaker relays the request to PATHMON only after the time-out expires. A task on the LINK queue is blocked from execution. It is removed from the LINK queue if PATHMON denies the request for a new link or if a link to the requested server class becomes available. A link can become available in two ways: an existing busy link becomes free, or a new link is created after the Speaker relays the link request to PATHMON.

Members of the DELINK queue are link entities. If the TCP decides to ask PATHMON to dissolve an existing link, it specifies a (possibly 0) time-out value for the link and places the link on the DELINK queue. The Speaker removes the link from the DELINK queue and relays the request to PATHMON only after the time-out expires. If, before the time-out expires, a terminal task requests a link that is of the same type as a link on the DELINK queue, the link is removed from the queue and the DELINK request cancelled.

Because members of the LINK and DELINK queues have time-out values associated with them, the Speaker does not honor LINK and DELINK requests on a first-in-first-out basis. The Speaker looks for an entity on the DELINK queue with an expired time-out value before it looks for a task on the LINK queue with an expired time-out value. Also, the Speaker completes a LINK or DELINK request before it starts another LINK or DELINK request.

The LINK entries under QUEUE INFO of the TCP statistics describe the LINK queue associated with the Speaker. LINK REQ CNT indicates the number of times tasks place themselves on the LINK queue. It does not indicate the number of SEND commands executed by the TCP. It also does not indicate how often link requests are sent to PATHMON because the task on the LINK queue may be taken off that queue before the Speaker relays its request to PATHMON.

LINK % WAIT indicates the percentage of requesting tasks finding others on the LINK queue at the time of the request. Because the Speaker does not honor a LINK request until the time-out expires, the % WAIT entry does not reflect the promptness with which the Speaker can handle LINK requests.

LINK MAX WAIT indicates the longest queue encountered by any task before it joins the LINK queue. LINK AVG WAIT indicates the average number of tasks on the queue.

The DELINK entries under QUEUE INFO of the TCP statistics describe the DELINK queue associated with the Speaker. REQ CNT indicates the number of times a link entity is placed on the DELINK queue. As with the REQ CNT for LINK requests, it does not indicate how often the DELINK requests are sent to PATHMON because the requesting entity may be taken off the DELINK queue before the Speaker relays its request to PATHMON.

DELINK % WAIT indicates the percentage of link entities finding others on the DELINK queue at the time of the request. Again, as with the LINK % WAIT figures, since the Speaker does not honor a DELINK request until the time-out expires, the % WAIT entry does not reflect the promptness with which the Speaker can handle DELINK requests.

DELINK MAX WAIT indicates the longest queue encountered by any link entity before it joins the queue for DELINK. DELINK AVG WAIT indicates the average number of requesting entities on the queue.

**Pools**

Each TCP maintains two pools of storage in memory: TERMPOOL and SERVERPOOL. Terminal tasks allocate and deallocate pool space dynamically. They request buffers from TERMPOOL for I/O operations to terminals and buffers from SERVERPOOL for I/O operations to servers.

A wait queue is associated with each pool. The TERMPOOL and SERVERPOOL entries under QUEUE INFO of the TCP statistics describe these queues. REQ CNT indicates the total number of times all terminal tasks requested a buffer allocation from the pool.

If no other task is on the wait queue and enough pool space is available to satisfy the request, the request is granted and the requesting task proceeds. Otherwise, the task joins the wait queue until other tasks return enough pool space to satisfy its request. While a task is on the wait queue it is blocked. The % WAIT entry indicates the percentage of tasks that were placed on the wait queue. MAX WAITS indicates the largest number of tasks on the wait queue after a task joins the queue. AVG WAITS indicates the average number of tasks on the wait queue.

The POOL INFO entries of the TCP statistics describe each of the two pools. SIZE is the maximum number of bytes available in the pool. REQ CNT is the number of requests for buffer allocation. MAX ALLOC is the largest number of bytes ever allocated at a given time. AVG ALLOC is the average size allocated. CUR ALLOC is the current allocated size. MAX REQ is the size of the largest single request for buffer allocation. AVG REQ is the average request size.

**Data Area**

The TCP allocates its data area in an extended segment. The DATA entries under AREA INFO of the TCP statistics describe the data area of the primary TCP process.

For each terminal task specified by the MAXTERMS parameter of the SET TCP command, the TCP allocates two slots of addresses in the data area. SLOT0 is approximately the size of MAXTERMDATA. The size of SLOT1 is approximately MAXTERMDATA + MAXREPLY. The sizes are approximate because the TCP rounds up to a page boundary (2048 bytes). The TCP uses SLOT0 of each task to simulate an execution stack for the executing Screen COBOL program units. This stack grows and shrinks during program execution. The TCP uses SLOT1 to hold the last checkpoint image.

The SIZE entry for AREA INFO indicates the total size of all slots for all terminal tasks. CUR ALLOC indicates the number of bytes used by all currently active tasks. MAX ALLOC indicates the largest number of bytes ever used within the data area.

**Code Area**

The TCP allocates its code area in the same extended segment as its data area. The CODE entries under AREA INFO of the TCP statistics describe the code area of the primary TCP process.

A compiled Screen COBOL program unit is composed of executable code and a sequence of screen descriptions. At run time, the executable code and the screen descriptions are fetched into the code area independently of each other. The screen descriptions are fetched when they are referenced, not when the program unit containing them is invoked.

Figure 3

```
TERM SERVTERM                                   16 DEC 1983, 10:18:46
  I/O INFO:      REQ CNT    MAX TSIZE    AVG TSIZE       I/O CNT
    DISPLAY            4          127          126             2
    ACCEPT             2           58           23             3
    SEND               1           22           22             1
    REPLY                          22           22
    CHECKPOINT                    616          359
  AREA INFO:    MAX SIZE     AVG SIZE     CUR SIZE
    DATA             636                       600
    CODE             648          334          584
```

**Figure 3.**
*A sample set of terminal statistics.*

When a terminal task is active, it can have up to three defined code segments: the executable code (Segment 0), the current base screen (Segment 1) and the current overlay screen (Segment 2). The current base screen is the screen used in the most recent DISPLAY BASE command. The current overlay screen is the screen used in the most recent operation affecting an overlay area of the current base screen. Segment 0 is always defined for an active task, and Segment 1 must be defined if Segment 2 is defined.

When a terminal task is about to execute, all its defined code segments must be present in the code area. When the task is not executing, its code segments can be overlaid by other code segments. A task's code segments are fetched and overlaid independently of each other; they need not be contiguous in the code area.

When a terminal task is about to execute, it requests the Code Manager to fetch all its defined segments. This counts as a single request in the REQ CNT entry of QUEUE INFO for MEMMAN. However, each defined segment that is fetched is counted separately in the REQ CNT entry of AREA INFO for CODE. The number of defined segments that are not present in the code area is used in the calculation of the AREA INFO % ABSENT entry for CODE.

For example, suppose a task with three defined segments is about to execute. This task requests the Code Manager to fetch three segments. For this request, REQ CNT for MEMMAN QUEUE INFO is 1. REQ CNT for AREA INFO is 3. Suppose two of the required segments are not in code area. The % ABSENT value for this request is then 67.

AREA INFO SIZE is the total number of bytes allocated to the code area. SIZE is specified by the CODEAREA parameter of the SET TCP command. MAX ALLOC indicates the largest number of bytes allocated from the code area at any time. AVG ALLOC indicates the average allocation. CUR ALLOC indicates the current allocation. MAX REQ indicates the largest sum of all code segments in a single request to the Code Manager. AVG REQ indicates the size of the average request.

## Terminal Statistics

Terminal statistics show the amount of total TCP resource used by a terminal task. A sample set of terminal statistics is shown in Figure 3. The I/O INFO entry indicates the number of I/O operations requested by the task. AREA INFO indicates the amount of code area and data area used by the task.

### I/O INFO

DISPLAY REQ CNT indicates the number of times the terminal task executes the following commands: DISPLAY BASE, DISPLAY OVERLAY, DISPLAY, RESET, TURN, CLEAR INPUT and SCROLL. When interpreting these commands, the TCP generates data for the terminal and places the data in a buffer in TERMPOOL. The TCP does not always output this buffer to the terminal immediately. When it requests an I/O operation to output the buffer, DISPLAY I/O CNT is incremented by 1. MAX TSIZE (maximum transfer size) indicates the size of the largest output buffer sent to the terminal in a single I/O operation. AVG TSIZE indicates the average size of the buffer for transmissions.

For block-mode terminals, the TCP outputs data to a terminal under the following conditions:

- When the buffer is full.
- When an ACCEPT is being interpreted.
- When certain other Screen COBOL commands (for example, the DELAY command) follow the commands that contribute to the request count (listed above).

For block-mode terminals, users specify the size of the output buffer allocated from TERMPOOL with the TERMBUF parameter of the SET TCP command. If the buffer is large enough, the TCP need never initiate a terminal output simply because a buffer is full. The minimum DISPLAY I/O CNT for these block-mode terminals is equal to ACCEPT REQ CNT.

Conversational-mode terminals initiate an output to the terminal whenever the DISPLAY command generates data for a screen line. The size of the terminal buffer is not determined by any user parameter.

ACCEPT REQ CNT indicates the number of ACCEPT commands executed by the task. For some block-mode terminals, the ACCEPT command generates two I/O operations: the first I/O reads the key, and the second I/O reads the data. Sometimes ACCEPT also displays ADVISORY messages or changes screen field attributes. I/O CNT indicates the total number of such I/O operations caused by ACCEPT commands. MAX TSIZE indicates the size of the largest transfer. AVG TSIZE indicates the size of the average transfer.

The SEND REQ CNT entry indicates the number of SEND commands that result in I/O operations to servers. I/O CNT indicates the number of I/O operations to servers. I/O CNT is always the same as REQ CNT for SEND commands. MAX TSIZE is the size of the largest transfer. AVG TSIZE is the size of the average transfer.

I/O INFO for REPLY and CHECKPOINT do not display REQ CNT and I/O CNT because there are no explicit requests for REPLY. Another reason for this is that the TCP performs more checkpoint operations than there are Screen COBOL checkpoint requests. For REPLY and CHECKPOINT, only MAX TSIZE and AVG TSIZE are displayed.

**AREA INFO**

The DATA entries are designed to help the user determine the best value for MAXTERMDATA. As described earlier, the TCP allocates two slots of data area in its extended segment for each terminal task. The size of SLOT0 is approximately MAXTERMDATA, and the size of SLOT1 is approximately MAXTERMDATA + MAXREPLY. The best MAXREPLY value can be determined from the MAX TSIZE entry for REPLY under I/O INFO. MAX SIZE for DATA under AREA INFO indicates the largest number of bytes used in the task's SLOT0. The largest MAX SIZE for all terminal tasks should be used to determine MAXTERMDATA. CUR SIZE indicates the current size used in SLOT0.

*Terminal statistics show the amount of total TCP resource used by a terminal task.*

MAX SIZE for CODE indicates the largest sum of all the code segments in a single request by the task. AVG SIZE indicates the size of the average request by the task. CUR SIZE indicates the size of the latest request by the task.

## Server Statistics

Server statistics describe the amount of total TCP resource used to communicate with a server class. In Figure 4, a set of sample TCP server statistics is shown.

**Figure 4.**
*A sample set of server statistics.*

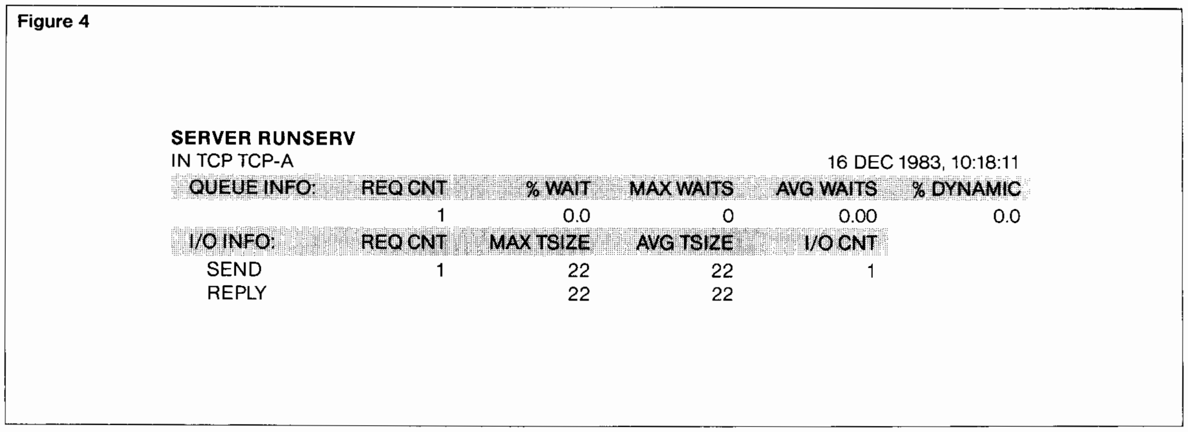


Figure 4

```
SERVER RUNSERV
IN TCP TCP-A                                                        16 DEC 1983, 10:18:11
  QUEUE INFO:    REQ CNT        % WAIT    MAX WAITS    AVG WAITS   % DYNAMIC
                       1           0.0            0         0.00         0.0
  I/O INFO:      REQ CNT     MAX TSIZE    AVG TSIZE      I/O CNT
    SEND               1            22           22            1
    REPLY                           22           22
```

**QUEUE INFO**
A link between the TCP and a server class can be either busy or available. It is busy when it is being used by a terminal task to communicate with a server; otherwise, it is available. Links become available in two ways: a busy link is freed by its user, or a new link to the server class is requested from and granted by PATHMON.

Before a terminal task can communicate with a server class, a link between the TCP and a server of that server class must be available. The terminal task acquires an available link at the beginning of a Screen COBOL SEND command. The link becomes available at the end of the SEND command.

When a terminal task executes a SEND command, the TCP checks to see if an existing link between the TCP and a server process of the requested server class is available. If an existing link is available, it is used; if not, the terminal task joins the wait queue associated with that server class. The terminal task is blocked while on this queue and is awakened when an available link is assigned to the task.

Before joining the wait queue for the server class, the terminal task may ask the Speaker to request a new link from PATHMON. The task asks for a new link if the TCP has enough resources to support one, i.e., if the number of links from the TCP to all server processes at the time of the request is less than that specified by the MAXSERVERPROCESSES parameter of the SET TCP command and if the number of server classes represented by processes that have links to the TCP is less than that specified by the MAXSERVERCLASSES parameter of the SET TCP command.

When a terminal task requests a new link, the TCP specifies a time-out value for the request and adds the task to the Speaker's LINK queue. The Speaker does not handle the request until after the time-out expires. The time-out value is set to either 0 or the value of the CREATEDELAY parameter of the SET SERVER command. It is set to the value of the CREATEDELAY parameter only if the number of existing links to the server class is greater than 0 and greater than the maximum number of static links available to the TCP for that server class. (Static links are defined below.) The maximum number of static links for that server class is initially set at the value of the NUMSTATIC parameter of the SET SERVER command. It can later be adjusted to lower values according to information passed back from PATHMON.

When a terminal task requests a new link, it joins both the Speaker's LINK queue and the server-class queue (see Figure 5). The task is awakened and taken off these queues when a link to the correct server class is available for it.

If PATHMON grants a new link, it designates the link as either static or dynamic. A static link is normally not dissolved unless an error occurs over that link. When a dynamic link becomes available and there are no terminal tasks waiting to use it, the TCP sets a time-out value for it (DELETEDELAY) and places it on the Speaker's DELINK queue. If the time-out expires, the Speaker sends the delink request to PATHMON, which dissolves the link.

The QUEUE INFO entries for server statistics describe the server-class wait queue. REQ CNT indicates the number of times a task joins the server-class wait queue. This represents the number of times a task does not find an available link immediately when it needs a link during a SEND command. The % WAIT entry indicates the percentage of requesting tasks that find other tasks on the wait queue at the time of their requests. MAX WAITS indicates the longest queue encountered by any task just before it joins the wait queue. AVG WAITS indicates the average queue length. The % DYNAMIC entry indicates the number of times a task asks for a link from the Speaker via the LINK queue with a time-out value of CREATEDELAY. It does not indicate the number of dynamic links granted by PATHMON because any request on the Speaker's LINK queue can be cancelled by the availability of another existing link.

**I/O INFO**

REQ CNT for SEND indicates the number of SEND commands that result in I/O operations to servers of this server class. I/O CNT indicates the number of I/O operations to servers of this server class. I/O CNT is always the same as the REQ CNT for SEND commands. MAX TSIZE is the size of the maximum transfer. AVG TSIZE is the size of the average transfer.

REQ CNT and I/O CNT for REPLY are not displayed because there are no explicit requests to REPLY.

## Conclusion

If understood properly, the statistics generated by the PATHWAY transaction processing system can be used to detect bottlenecks within the TCP process. The % WAIT and % ABSENT entries are especially useful for this. Changing the configuration of the PATHWAY system based on careful analysis of these statistics can help to eliminate some performance degradations.

Figure 5

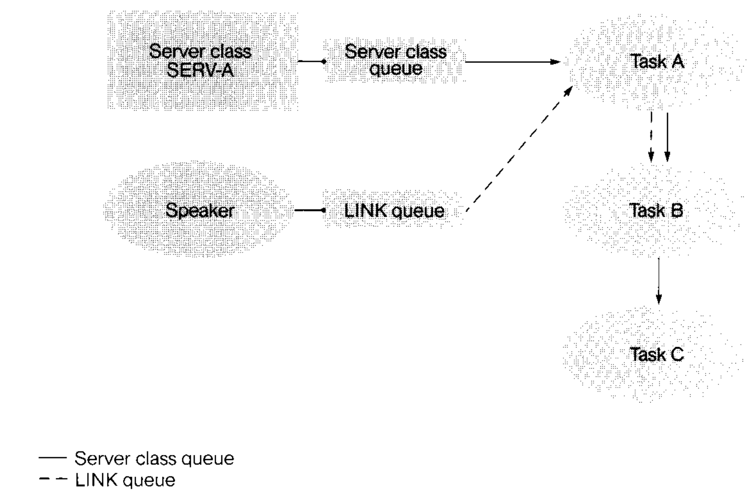


**Figure 5.**
*Tasks waiting for a link to a server class may be on the server class queue only or on both the server class queue and the LINK queue. In this example, tasks A, B, and C are waiting for an available link. Tasks A and B are also trying to obtain a new link from the PATHWAY Monitor via the Speaker.*

Other performance factors affecting the PATHWAY system are not represented in these statistics. These factors generally involve the way the TCP contends for system resources with other processes that are part of the GUARDIAN operating system or with other parts of the PATHWAY system. Some examples are: paging by GUARDIAN incurred by the TCP, the ability of the TCP to handle I/O terminations promptly, and the ability of PATHMON to grant links promptly. Consequently, in tuning a system that uses the PATHWAY transaction processing system, it is essential to analyze the statistics produced by the GUARDIAN operating system together with those produced by the PATHWAY system.

**Raymond Wong**, of Tandem's Software Development Group, wrote this article and the preceeding one, "A New Design for the PATHWAY TCP."

# A SNAX Passthrough Tutorial

SNA Communications Services (SNAX™) is the latest in the series of gateway products from Tandem. It allows Tandem customers to gain access to, and share resources with, an IBM Systems Network Architecture (SNA) environment. Essentially SNAX acts as a subhost to an IBM SNA mainframe, supporting SNA physical and logical units (PUs and LUs) and providing SNA PU and LU emulation. In conjunction with these subhost features, SNAX allows data to pass between SNA applications and SNA LUs through a Tandem system (or systems connected via an EXPAND™ network) transparently. With this capability, known as SNA Session Passthrough, a Tandem system or EXPAND network can be integrated into an existing SNA network to provide an interface that is operationally and functionally compatible with existing SNA terminal use. Not only does this provide a clean, easy migration path to the Tandem environment, it affords a degree of flexibility that was not previously possible with "traditional" SNA networks.

The following discussion is intended for analysts, systems programmers, and system managers who design and implement SNA and SNA/EXPAND networks, and who wish to take advantage of SNAX Passthrough capabilities. The major passthrough considerations addressed here include:

- Capabilities.
- Implementation.
- IBM generation.
- Tandem generation.
- Activation.
- Logon/Logoff.

## Passthrough Capabilities

SNAX Passthrough allows SNA devices to be attached to a Tandem NonStop II or NonStop TXP system (or several Tandem systems connected via an EXPAND network) and to access SNA application programs that are resident in any IBM host, without change or reprogramming. SNAX imposes no restrictions on the SNA applications that can participate in passthrough sessions; however, some SNA applications support only some types of SNA device. Passthrough does not change this. Examples of SNA applications that can participate in passthrough sessions include:

- Information Management System (IMS).
- Customer Information Control System (CICS).
- Time Sharing Option (TSO).
- Job Entry Subsystem 2 (JES2).
- Job Entry Subsystem 3 (JES3).
- Network Communications Control Facility (NCCF).
- Host Command Facility (HCF).

Passthrough is applicable to the PU Type 2 (PU.T2) only. This category includes:

- IBM 3274 Information Display System.
- IBM 3276 Information Display System.
- IBM 3600 Financial System.
- IBM 3624 Financial System.
- IBM 3630 Plant Communications System.
- IBM 3640 Manufacturing System.
- IBM 3650 Retail Store System.
- IBM 3660 Supermarket System.
- IBM 3680 Programmable Store System.
- IBM 3770 Data Entry System.
- IBM 4700 Financial System.
- IBM 8100 Information System.
- IBM Series/1 General Purpose DP System.
- IBM System/32 General Purpose DP System.
- IBM System/34 General Purpose DP System.
- IBM System/38 General Purpose DP System.

Specifically not supported in passthrough sessions is the PU Type 1 (PU.T1). This category includes:

- IBM 3271 Display System.
- IBM 3767 Communications Terminal.
- IBM 5250 Information Display System.
- IBM 6670 Information Distributor.

Although many different types of PU Type 2 can be supported by Passthrough, for the sake of simplicity, all examples and configurations in this article refer to the IBM 3274 Information Display System as the PU, and the IBM 3278 Display Unit as the LU.

To illustrate what SNAX Passthrough is and where it fits into an existing network, Figures 1 and 2 show how a Tandem Non-Stop II or NonStop TXP can be integrated into a traditional SNA environment. Figure 1 shows a Passthrough configuration in which a single Tandem node supports both the connections to the SNA devices and the connections to the SNA host. In this configuration the EXPAND networking software is not required.

Figure 2 shows a Passthrough configuration in which SNA devices are dispersed geographically throughout a Tandem EXPAND network; i.e., the SNA devices are not attached to the same Tandem node as the SNA host.

Figure 1

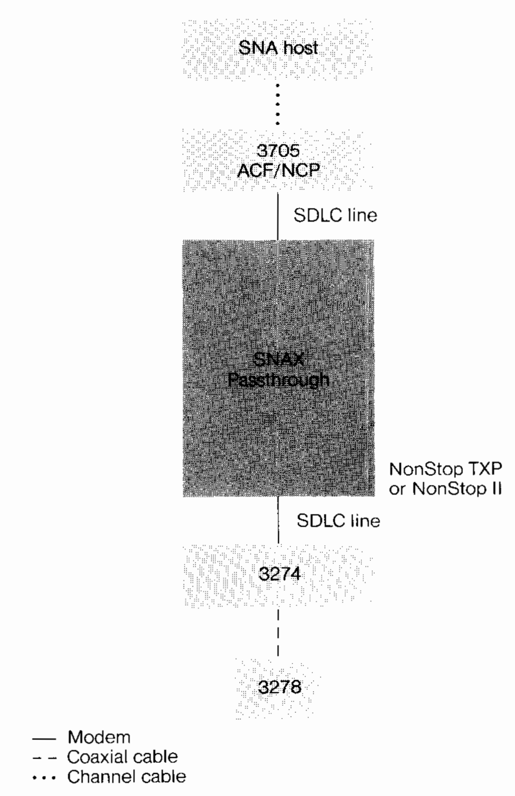


**Figure 1.**
*Passthrough can be configured into a single Tandem system with SNAX. In this configuration the same Tandem system connects the IBM host and the SNA devices.*

Figure 2

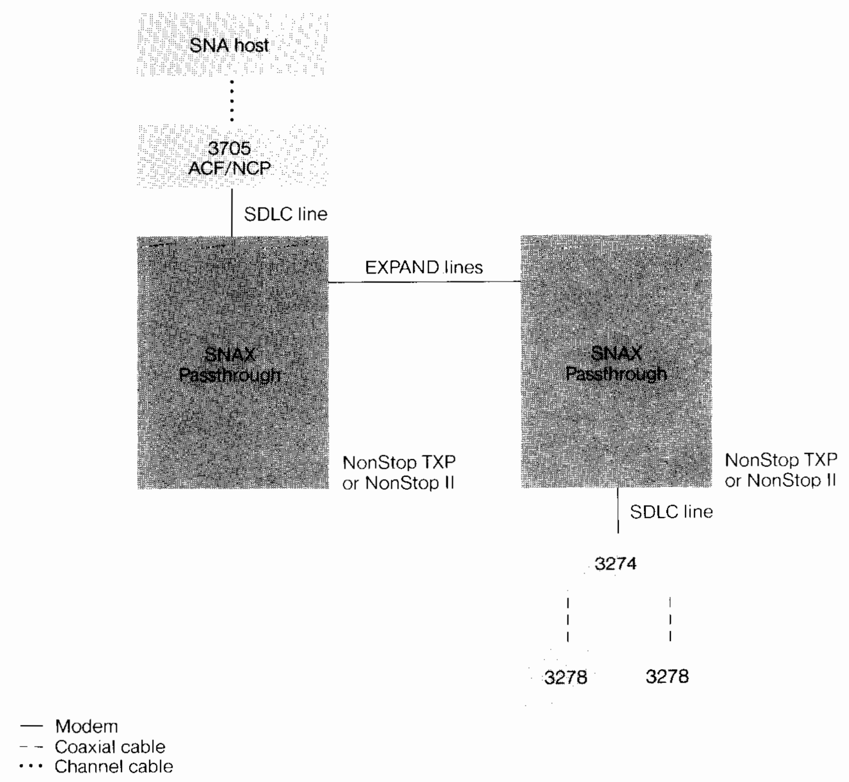


**Figure 2.**
*Passthrough can be configured into multiple Tandem systems with SNAX. In this configuration one Tandem system connects the IBM host, and another Tandem system connected via an EXPAND network connects the SNA devices.*

Figure 3

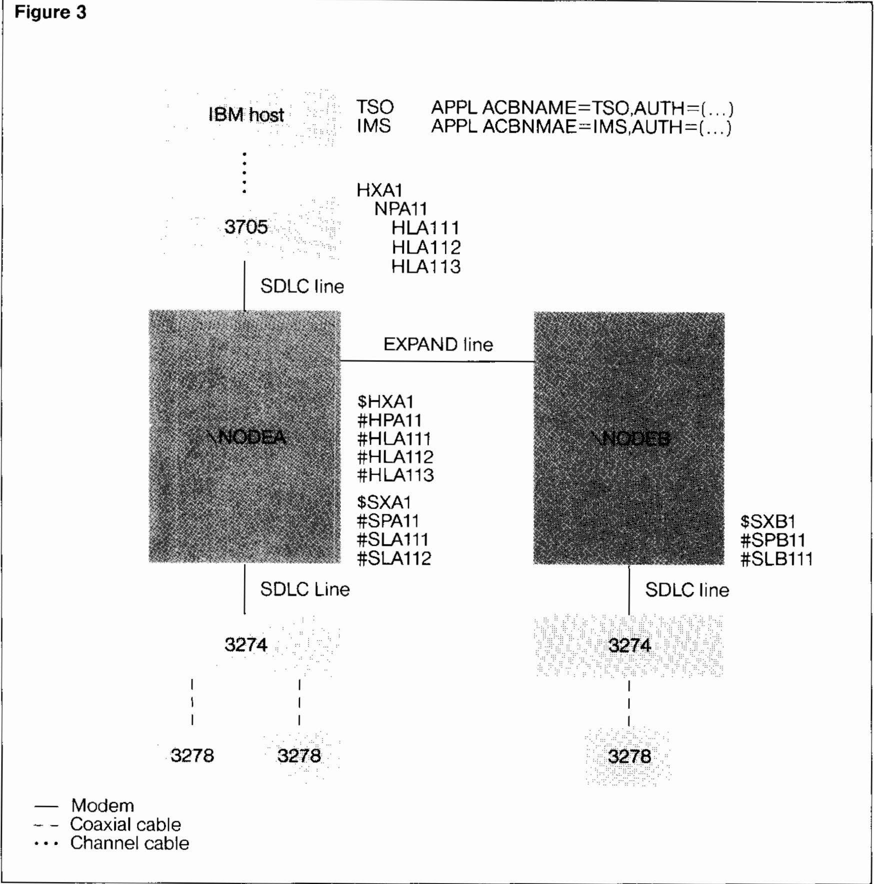


**Figure 3.**
*A SNAX Passthrough configuration in which the SNA devices are attached to various nodes throughout the EXPAND network. Although this is a simple example, it illustrates all the elements of passthrough.*

With either of these configurations SNAX provides two modes of passthrough operation, *dynamic* mode and *associated* mode. Each has particular attributes that lend themselves to specific applications and solutions.

Dynamic passthrough allows the mapping between a specific SNA device attached to a Tandem EXPAND network and a corresponding passthrough LU defined on the IBM host to be established by SNAX in a dynamic manner. SNAX uses a *next-available* algorithm to determine this. This mode of operation is particularly suitable in environments that:

1. Map a large number of SNA devices to a small(er) number of LUs defined to the IBM host.
2. Require no particular association between the SNA device and the LU defined to the IBM host.

Associated passthrough establishes a one-to-one mapping between a specific SNA device attached to a Tandem EXPAND network and a particular passthrough LU defined to the IBM host. This mapping is specified by the ASSOCIATE ‹LU name› parameter, to be defined as part of the configuration process.[1]

This mode of operation is particularly suitable in environments that:

1. Map the SNA devices on a one-to-one basis to the LUs defined to the IBM host.
2. Require a fixed association between the SNA device and the LU defined to the IBM host. This association is necessary if the host application initiates the session (i.e., the SNA application *acquires* the terminal), or if security and authorization parameters are defined by the IBM application for each device.

## Passthrough Implementation

Passthrough is a function of the SNAX line-handler software. Its implementation can be considered as a logical software connection between the SDLC communications lines. Once a Passthrough session has been established by SNAX, all data flows through the Tandem EXPAND network transparently. SNAX does not act upon or process the data that is passed through; it merely monitors it for session termination requests. A session termination request is indicated by receipt of an UNBIND command from the SNA application program.

This implementation has distinct advantages over other possible passthrough methods:

▪ The processing required to perform the passthrough is reduced to the absolute minimum within SNAX. This ensures that passthrough is completely transparent, while minimizing any impact to the performance characteristics on the system.

▪ Because Passthrough passes not only all LU-to-LU session data, but all SNA commands between the SNA device and the SNA application, SNAX is not sensitive to changes. IBM host SNA software, applications, and SNA devices can be implemented, modified, and upgraded without affecting SNAX.

[1]Uppercase characters represent keywords and reserved words. Lowercase characters enclosed in angle brackets (‹›) represent variable entries supplied by the user.

The logical configuration, i.e., the configuration that is presented to and viewed by the IBM host, is the same in both associated and dynamic modes of operation. The IBM host views the Tandem nodes as a PU Type 2 (PU.T2) with a number of attached LUs. These are passthrough LUs that are configured into the SNAX environment. They have no relationship with the device LUs that may be dispersed throughout the Tandem EXPAND network; however, before Passthrough can become operational, an association has to be established between these passthrough LUs and the SNA device in the network.

That association is established as part of the LOGON request for the terminal operator. In associated passthrough mode this association is established to the specific passthrough LU that was configured with the ASSOCIATE ‹LU name› parameter. In dynamic passthrough mode, SNAX establishes this association to the next available passthrough LU that does not have an ASSOCIATE ‹LU name› parameter.

## IBM Configuration

Figure 3 contains the example configuration used in the remainder of this discussion. It shows an IBM SNA host running SNA applications (TSO and IMS), accessed through an SNA communications controller. This could be an IBM 3705, 3725, or equivalent, with ACF/NCP. This SNA host environment is connected to a Tandem EXPAND network via an SDLC communications line. The SNA controllers and devices are distributed throughout the Tandem EXPAND network. In this example configuration, the element names (Line, PUs, and LUs) have been chosen to reflect logical associations between corresponding elements; e.g., #HLA111 is the Tandem host LU that is equivalent to the IBM definition HLA111. SNAX does not require this correspondence; it is used as a convention in this article only. Figure 4 shows the naming convention used for the elements in this configuration. It is used throughout the rest of the article.

The basis for an ACF/NCP Stage I generation for this configuration is shown in Figure 5.

Figure 4

Example element names

| Element class | Element type | Unique NODE identifier | Unique LINE identifier | Unique PU identifier | Unique LU identifier | |
|---|---|---|---|---|---|---|
| H | X | A | 1 | | | Host Line 1 from Node A |
| H | P | A | 1 | 1 | | Host PU 1 on Line 1 from Node A |
| H | L | A | 1 | 1 | 1 | Host LU 1 on PU 1 on Line 1 from Node A |
| H | L | A | 1 | 1 | 2 | Host LU 2 on PU 1 on Line 1 from Node A |
| H | L | A | 1 | 1 | 3 | Host LU 3 on PU 1 on Line 1 from Node A |
| S | X | A | 1 | | | SNAX Line 1 from Node A |
| S | X | B | 1 | | | SNAX Line 1 from Node B |
| S | P | A | 1 | 1 | | SNAX PU 1 on Line 1 from Node A |
| S | P | B | 1 | 1 | | SNAX PU 1 on Line 1 from Node B |
| S | L | A | 1 | 1 | 1 | SNAX LU 1 on PU 1 on Line 1 from Node A |
| S | L | A | 1 | 1 | 2 | SNAX LU 2 on PU 1 on Line 1 from Node A |
| S | L | B | 1 | 1 | 1 | SNAX LU 1 on PU 1 on Line 1 from Node B |

**Element classes:**
H host
S SNAX

**Element types:**
N node
X line
P PU
L LU

Figure 5

```
               GROUP  LNCTL=SDLC,DIAL=NO,TYPE=NCP,
                      REPLYTO=1.0,
                      TEXTTO=3.0
      HXA1     LINE   ADDRESS=020,       Single 3705 port address
                      SPEED=9600,        Required for 3705 only
                      CLOCKING=EXT,      Clocking by modem
                      DUPLEX=FULL,       Modem Option for RTS/CTS
                      NEWSYNC=NO,        Modem Option for multipoint line
                      NRZI=NO,           Modem Option for transmission mode
                      POLLED=YES
               SERVICE ORDER=HPA11
      HPA11    PU     ADDR=C1            SDLC address in hexidecimal
                      PUTYPE=2,          PU.T2
                      DATMODE=HALF,      Alternate SEND/RECEIVE
                      ISTATUS=INACTIVE,  Initially inactive
                      MODETAB=MTPU2,     Mode table
                      USSTAB=USSPU2,     USS table
                      SSCPFM-USSSCS,     SSCP-LU data
                      MAXOUT=7,          Outstanding SDLC frames
                      MAXDATA=265        Maximum PIU data
      HLA111   LU     LOCADDR=1,BATCH=NO
      HLA112   LU     LOCADDR=2,BATCH=NO
      HLA113   LU     LOCADDR=3,BATCH=NO
```

**Figure 4.**
*The naming convention used in the example configuration illustrated in Figure 3. The convention is used throughout the remainder of this article. It provides as much information as possible about the type of element identified (e.g., from its name, the characteristics and location of the element can be easily determined).*

**Figure 5.**
*The basis for an ACF/NCP Stage I generation for the example configuration in Figure 3.*

Figure 6

```
\NODEA SDLC communications line to the IBM host:
$HXA1    BITA.0,BITA.1   SNATS   TYPE 58, SUBTYPE 0,
                                 DRIVER SNA^6203^DVR,
                                 INTERRUPT SNA^6203^INTERRUPT,
                                 L2PROTOCOL SNATS^LV2^PROTOCOL,
                                 L3PROTOCOL SNATS^LV3^PROTOCOL,
                                 L4PROTOCOL SNATS^LV4^PROTOCOL,
                                 SMLDEV $SSCP, RSIZE 1000,
                                 LOCALPOOLPAGES 60,
                                 L6PROTOCOL SNATS^SNALU,
                                 FULL,
                                 SECONDARY,
                                 ALLADDRRD,
                                 FLAGFILL,
                                 CONTCF,
                                 FRAMESIZE 267,
                                 L2WINDOW 7,
                                 LINEBUFFERSIZE 4096,
                                 MAXPUS 1,
                                 MAXLUS 3;

\NODEA SDLC communications line to the IBM 3274 Controller:
$SXA1    BITA.0,BITA.1   SNATS   TYPE 58, SUBTYPE 0,
                                 DRIVER SNA^6203^DVR,
                                 INTERRUPT SNA^6203^INTERRUPT,
                                 L2PROTOCOL SNATS^LV2^PROTOCOL,
                                 L3PROTOCOL SNATS^LV3^PROTOCOL,
                                 L4PROTOCOL SNATS^LV4^PROTOCOL,
                                 SMLDEV $SSCP, RSIZE 1000,
                                 LOCALPOOLPAGES 60,
                                 L6PROTOCOL SNATS^SNALU,
                                 FULL,
                                 PRIMARY,
                                 FRAMESIZE 267,
                                 L2WINDOW 7,
                                 LINEBUFFERSIZE 4096,
                                 MAXPUS 1
                                 MAXLUS 2;

\NODEB SDLC communications line to the IBM 3274 Controller:
$SXB1    BITA.0,BITA.1   SNATS   TYPE 58, SUBTYPE 0,
                                 DRIVER SNA^6203^DVR,
                                 INTERRUPT SNA^6203^INTERRUPT,
                                 L2PROTOCOL SNATS^LV2^PROTOCOL,
                                 L3PROTOCOL SNATS^LV3^PROTOCOL,
                                 L4PROTOCOL SNATS^LV4^PROTOCOL,
                                 SMLDEV $SSCP, RSIZE 1000,
                                 LOCALPOOLPAGES 60,
                                 L6PROTOCOL SNATS^SNALU,
                                 FULL,
                                 PRIMARY,
                                 FRAMESIZE 267,
                                 L2WINDOW 7,
                                 LINEBUFFERSIZE 4096,
                                 MAXPUS 1,
                                 MAXLUS 1;
```

**Figure 6.**
*Definition of the communications lines in the SYSGEN configuration for the example configuration (assuming both \NODEA and \NODEB are configured with a 6203 Bit-synchronous Controller that has been generated with the user-specified name of BITA). The shaded items are standard parameters for all SNAX lines and should be included as shown.*

## Tandem Configuration

In the Tandem nodes, the only elements that must be generated are the physical communications lines. Assuming that both \NODEA and \NODEB are each configured with a 6203 Bit-synchronous Controller that has been generated with the user-specified name of *BITA*, the communications lines are defined in the system generation (SYSGEN) configuration files as shown in Figure 6. The items in boldface type are standard parameters for all SNAX lines and should be included as shown.

Those parameters that are not standard for all SNAX lines are explained below. For those corresponding to ACF/NCP generation parameters, the ACF/NCP parameter is given. Care must be taken to ensure that the parameter values correspond.

*L6PROTOCOL3 SNATS^SNALU* indicates that the SNALU interface is to be used to access this line. All host communications lines require SNALU.

*FULL* indicates that the communications facility can provide a full duplex channel for data communications, and SNAX is to assume that Clear To Send (CTS) is always present at the modem interface. This is only possible when SNAX is the only PU on the line. If SNAX is multidropped with other PUs, specify HALF. The corresponding parameter in the ACF/NCP generation is DUPLEX=FULL on the LINE macro.

*SECONDARY* indicates that this Tandem node is to assume the role of secondary SDLC station for this communications line. (When communicating with a PU.T2, ACF/NCP always assumes the role of primary station).

*ALLADDRRD* indicates that this Tandem node is to accept SDLC frames for all SDLC addresses.

*FLAGFILL* indicates that the 6203 Bit-synchronous Controller is to transmit a constant stream of SDLC flag characters (X'7E') when no other data is present.

*CONTCF* indicates that the Receive Line Signal Detector (EIA circuit CF, CCITT circuit 109) is strapped ON, for support of constant carrier modems.

*FRAMESIZE 267* indicates the maximum size of the SDLC frame that can be transmitted on the line. This value is comprised of maximum PIU size (usually 265 bytes) and the SDLC framing characters (A and C fields, always 2 bytes).

*L2WINDOW 7* indicates the maximum number of SDLC frames that can be transmitted on the line by the primary station without a response. This number is generally in the range of one to seven. The corresponding parameter in the ACF/NCP generation is MAXOUT=7 on the PU macro.

*LINEBUFFERSIZE 4096* indicates the buffer size required for line operations. It is comprised of the L2WINDOW x FRAMESIZE x 2.

*MAXPUS 1* indicates the maximum number of PUs that can be defined on the line. (This number must be greater than or equal to the number of PU macros defined in the ACF/NCP generation for this line *or* the number of PUs that can be added by ACF/NCP dynamic reconfiguration.)

*MAXLUS 3* indicates the maximum number of LUs that can be defined on the line. (This number must be greater than or equal to the number of LU macros defined in the ACF/NCP generation for all PUs on this line *or* the number of LUs that can be added by ACF/NCP dynamic reconfiguration).

*PRIMARY* indicates that this Tandem node is to assume the role of primary SDLC station for this communications line. (When communicating with ACF/NCP, a PU.T2 always assumes the role of secondary station).

Having defined the communications lines ($HXA1, $SXA1 and $SXB1) to nodes \NODEA and \NODEB, it is only necessary to dynamically configure the environment on those lines. Dynamic configuration is available through the operation of the Tandem Configuration Management Interface utility (CMI). The configuration can be dynamically added, modified, or deleted as required. The CMI commands to configure the environment in Figure 3 follow below. The items in boldface type are standard parameters for all SNAX lines and should be included as shown.

To define the SNA devices to SNAX on \NODEA, enter the CMI commands:

```
ADD LINE $SXA1

ADD PU  $SXA1.#SPA11,      TYPE(13,2),
                           RECSIZE 265,
                           ADDRESS 199

ADD LU  $SXA1.#SLA111,     TYPE(14,0),
                           PUNAME #SPA11,
                           RECSIZE 1024,
                           PROTOCOL SNALU,
                           ADDRESS 1

ADD LU  $SXA1.#SLA112,     LIKE $SXA1.#SLA111,
                           ADDRESS 2
```

To define the SNA devices to SNAX on \NODEB, enter the CMI commands:

```
ADD LINE $SXB1

ADD PU  $SXB1.#SPB11,      TYPE(13,2),
                           RECSIZE 265,
                           ADDRESS 199

ADD LU  $SXB1.#SLB111,     TYPE(14,0),
                           PUNAME #SPB11,
                           RECSIZE 1024,
                           PROTOCOL SNALU,
                           ADDRESS 1
```

To define the environment with which the IBM host communicates, enter the following CMI commands on \NODEA. Note that in this configuration, all the passthrough LUs (#HLA111,#HLA112,#HLA113) are available for dynamic allocation.

```
ADD LINE $HXA1,            APPLID(TSO,IMS)

ADD PU  $HXA1.#HPA11,      TYPE(13,2),
                           RECSIZE 265,
                           ADDRESS 193

ADD LU $HXA1.#HLA111,      TYPE(14,0),
                           PUNAME #GPA11,
                           RECSIZE 1024,
                           PROTOCOL SNALU,
                           PASSTHROUGH ON,
                           ADDRESS 1

ADD LU  $HXA1.#HLA112,     LIKE $HXA1.#HLA111,
                           ADDRESS 2

ADD LU  $HXA1.#HLA113,     LIKE $HXA1.#HLA111,
                           ADDRESS 3
```

The parameters that are not standard for all SNAX lines are explained below. For those that correspond to ACF/NCP generation parameters, the ACF/NCP parameter is given. Again, care must be taken to ensure that the parameter values for these correspond.

*RECSIZE 265* (on ADD PU) indicates the maximum PIU size that can be transmitted to this PU. This value is comprised of the PU buffer size (usually 256 bytes), the RH (always 3 bytes), and the FID2 TH (always 6 bytes). The corresponding parameter in the ACF/NCP generation is MAXDATA=265 on the PU macro.

*ADDRESS 193* (on ADD PU) indicates the line address of the PU that is used as the ADDRESS field in the SDLC frame. This is a decimal equivalent of X'C1'. The corresponding parameter in the ACF/NCP generation is ADDR=C1 on the PU macro.

*RECSIZE 1024* (on ADD LU) indicates the maximum RU size that can be transmitted to this LU. This value depends upon the session parameters and is specified in the BIND command.

*ADDRESS 1* (on ADD LU) indicates the local address on the PU that provides the support for this LU. The corresponding parameter in the ACF/NCP generation is LOCADDR=1 on the LU macro.

*PASSTHROUGH ON* indicates that this is a passthrough LU.

## Passthrough Activation

In order to enable Passthrough operation, the configuration should be activated in a predefined sequence. This is necessary in order to satisfy the hierarchical manner in which SNA expects activation to occur. If this sequence is not maintained, it is possible to recover the environment, provided that strict error recovery procedures are enforced at the IBM host.

Basically, all Tandem elements should be activated before the equivalent IBM elements. This ensures that the environment is operational before the SNA host attempts activation. The suggested order is:

1. From the Tandem host(s), activate the SNA devices.
2. From the Tandem host(s), activate the environment with which the IBM host communicates.
3. From the IBM host(s), activate the environment with which the IBM host communicates.

For the configuration example in Figure 3, the sequence of commands is the following. To activate the SNA devices on \NODEA, enter these CMI commands.

```
START LINE   $SXA1
START PU     $SXA1,#SPA11
START LU     $SXA1,#SLA111
START LU     $SXA1,#SLA112
```

To activate the SNA devices on \NODEB, enter these CMI commands:

```
START LINE   $SXB1
START PU     $SXB1,#SPB11
START LU     $SXB1,#SLB111
```

To activate the environment with which the IBM host communicates, enter these CMI commands on \NODEA:

```
START LINE   $HXA1
START PU     $HXA1,#HPA11
START LU     $HXA1,#HLA111
START LU     $HXA1,#HLA112
START LU     $HXA1,#HLA113
```

To activate the environment with which the IBM host communicates, enter these VTAM commands on the IBM host:

```
VARY NET, ACT, ID=HXA1
VARY NET, ACT, ID=HPA11
VARY NET, ACT, ID=HLA111
VARY NET, ACT, ID=HLA112
VARY NET, ACT, ID=HLA113
```

## Passthrough Logon

Once the environment has been activated, the terminal operator can establish a passthrough session by requesting a LOGON to a particular IBM host application. This is done by depressing the SYSREQ key to enter SSCP-LU session and then typing in the LOGON command.

The LOGON command takes one of two forms, depending upon the location of the requesting terminal. For terminals attached to the same node as the requested IBM host (e.g., if #SLA111 wanted to pass through #HXA1 to IMS), the command would be LOGON #HXA1.IMS. For terminals attached to a node other than the requested IBM host (e.g., if #SLB111 wanted to pass through #HXA1 to TSO), the command would be LOGON \NODEA.$HXA1.TSO.

In both instances, since dynamic passthrough is being used, the LOGON request maps the SNA device to the next available passthrough LU (i.e., $SLA111 is mapped to #HLA111 and #SLB111 is mapped to #HLA112.) The associations are established by SNAX with INITSELF commands to the IBM host application. The associations stay in effect until the passthrough session is terminated, and at this time, the passthrough LU is free for another dynamic passthrough session.

The general format of the LOGON command is:

```
LOGON (<node name.>) <line name>.<appl name>
```

In associated passthrough mode, the passthrough LU can be mapped to only one SNA device. This mapping is determined when the passthrough LU is configured. The appropriate CMI commands to define this type of configuration would be:

```
ALTER LU $HLA112,
  ASSOCIATE \NODEA.$SXA1.#SLA111

ALTER LU $HLA113,
  ASSOCIATE \NODEB.$SXB1.#SLB111
```

This command defines the mapping that is allowed; it does not establish the association. The association can only be established by the LOGON command from the terminal. With this configuration, the terminal operator would LOGON in exactly the same manner, but the association would be somewhat different; i.e., #SLA111 would be mapped to #HLA112, and #SLB111 would be mapped to #HLA113. The association stays in effect until the session is terminated, and the mapping stays in effect until another ALTER LU command is entered.

## Passthrough Logoff

In both modes of operation, both the IBM host application and the terminal user have the ability to terminate the passthrough session. Ultimately, the application (or the SNA software, on behalf of the application) has the responsibility of terminating the session by issuing the SNA command UNBIND; however, the terminal operator can request this action by requesting LOGOFF. The LOGOFF request can take one of three forms:

1. A character-coded request that the application recognizes and acts upon to issue the UNBIND. This method of LOGOFF is product-specific, and details for its implementation should be obtained from the relevant product documentation.
2. A character-coded request that SNAX recognizes and acts upon. When SNAX detects the LOGOFF command, it formats and issues the SNA command TERMSELF to the IBM host, which in turn notifies the application. The application then issues the UNBIND.
3. A TERMSELF command that SNAX forwards to the IBM host. The latter then notifies the application, which in turn issues the UNBIND.

## Conclusion

With SNAX, a Tandem system or EXPAND network can be integrated into an existing SNA network. With SNAX Passthrough, SNA data can pass through a Tandem system transparently. By following the steps described in this article, SNA users can implement SNAX Passthrough to connect SNA products to Tandem NonStop II or NonStop TXP systems, thus benefiting from the flexibility and reliability of Tandem systems.

**David Kirk** is Tandem's product manager for SNAX. He has been deeply involved with SNA implementation since its introduction in 1974, in both support and consultation roles.

# The TRANSFER Delivery System For Distributed Applications

Recent enhancements to the ENCOMPASS™ distributed data base system and the introduction of the TRANSFER™ delivery system have made it easier to (a) distribute online transaction processing (OLTP) applications on Tandem systems, (b) couple diverse applications loosely, and (c) create gateways into the system for devices produced by other manufacturers:

- The PATHWAY transaction processing system now allows PATHWAY Monitors (PATHMONs) on several nodes to coordinate the assignment of links to servers. Thus, a requester on one node can access a server class on another and in that way distribute the processing load.
- The Transaction Monitoring Facility (TMF) has been modified so that it can protect network transactions. Thus, a program can now update files on several nodes as part of a single TMF transaction.
- The new TRANSFER delivery system provides timed information delivery and "as-soon-as-possible" network transport of data for distributed applications, Tandem devices, and other devices.

Both ENCOMPASS and TRANSFER help the application designer to implement distributed applications efficiently. However, each was designed to meet different needs. The ENCOMPASS system is best suited for performing *tightly-coupled* operations (e.g., multiple record updates required to process a complicated transaction in a single distributed OLTP application), while the TRANSFER system was designed to facilitate the exchange of information among *loosely-coupled* applications and devices.

## Tight Coupling of Operations in a Distributed OLTP Application

An on-line transaction processing (OLTP) application is one in which data relevant to a specific set of business transactions is processed and recorded as each transaction occurs. Because customers are often waiting (e.g., for confirmation that a reservation has been made or that a specific seat has been assigned), each unit of work (each transaction) must be processed within 3 to 10 seconds, even though a complex exchange of messages updating several different data files on several different network nodes may be required. If all operations associated with that unit of work cannot be accomplished within that time, consistency dictates that all updates that have been completed must be backed out (by TMF in the Tandem environment) and the unit of work handled at some later time.

For example, consider an airline reservation system in which seat assignments are made at the same time as the reservation. If the application is distributed over a Tandem network, all collaborating systems must have some way of learning about the most recent seat assignment (so as to avoid assigning that seat to someone else):

▪ If files are duplicated throughout the network, a requester process on the terminal's node has to communicate with a server process on each node where such a file resides so that each file can be updated.

▪ If the data base is on one node, but many nodes support terminals that use the application, a requester process on the terminal's node must be able to communicate with one or more servers on the data base node to update the appropriate records.

These are all tightly-coupled operations. That is, they must all be performed in the 3- to 10-second period of time referred to above, or the data base must be returned to its original state. The new enhancements to PATHWAY and TMF allow requesters on the terminal's node to link to servers on all remote nodes and to perform any required data base updates as part of a single TMF transaction. TMF guarantees that if all the required operations are not completed, those that were are backed out.

## Loose Coupling Among Distributed OLTP Applications

Figure 1 shows a collection of OLTP applications serving several different organizations within a company. These applications are loosely coupled. That is, they communicate with one another regarding the transactions that they process, but that communication does not need to take place in the 3 to 10 seconds allotted for processing the transaction. If, for some reason, the information cannot be forwarded immediately, the transaction can be completed and the relevant information passed on at some later time. For example, the order-entry application in Figure 1 provides input for the credit and order validation applications, but if that input is provided a few minutes or hours after the order has been entered, nothing of significance has been lost. The reliability of the delivery mechanism is far more important than its speed.

In a Tandem environment, the TRANSFER delivery system provides precisely the kind of mechanism needed for information exchange among the applications shown in Figure 1. Thanks to TRANSFER, loosely-coupled applications are not forced to use the ENCOMPASS system, which requires tight coupling of all operations associated with a transaction. Moreover, TRANSFER does not force these applications to revert to a batch mode of information exchange either (i.e., users do not have to wait until the end of the day or week to send information to or receive information from another application). TRANSFER works independently of the original input and delivers packages of data reliably, one at a time, exactly once, and as soon as possible.

Figure 1

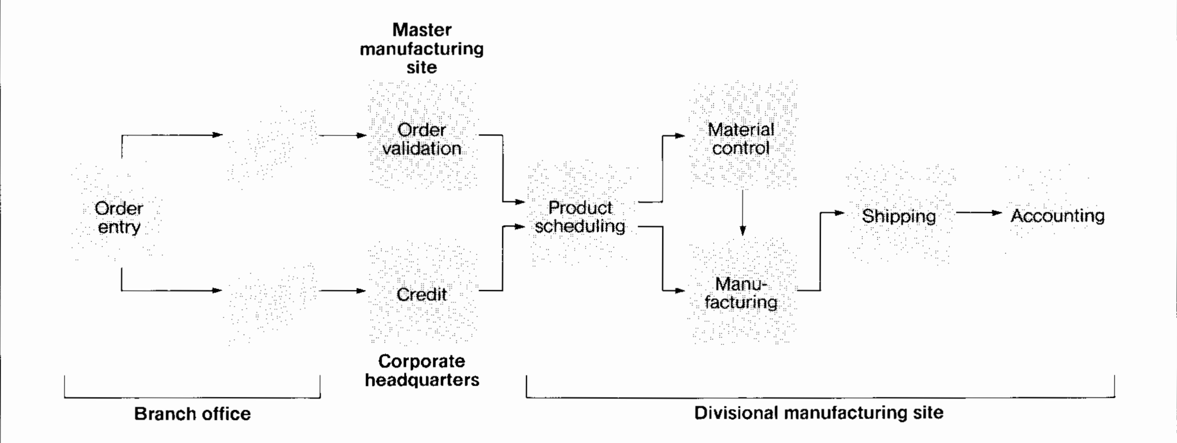


**Figure 1.**
*Applications in a typical manufacturing company. In this type of data processing environment, the delivery of paper has traditionally triggered the next processing step in a series of "loosely-coupled" applications. Now the TRANSFER delivery system assumes responsibility for delivering the information reliably and as soon as possible.*

When all the applications that make up an information system are running on one local computer, it makes little difference whether the system is implemented as one application (with tight coupling of all operations associated with a given transaction) or as several loosely-coupled applications. However, when the applications are distributed over a network, as in Figure 1, it makes a very big difference. Consider, for example, what happens if all operations are tightly coupled and a communications line to one of the nodes goes down. The system will not be able to communicate with that node, and therefore, it will not be able to complete any transaction that requires READ or WRITE access to that node. Since a TMF transaction must be complete to be successful, the application has only three alternatives. It can:

*For each correspondent, TRANSFER defines a "depot" or private storage areas for a user's data and information.*

- Retry until it is successful. (There is no way to know how long this could take.)
- Back out the transaction and have the operator re-enter it later.
- Subdivide the transaction into separate transactions, one for each node. (The program is then responsible for maintaining data base consistency and for retrying each node until it is successful.)

The last alternative is more difficult than it sounds. Exception handling is complex to control programmatically and becomes more difficult the more nodes there are.

In a distributed system made up of loosely-coupled applications, each application stands alone, processing information whenever it arrives from the previous application in the sequence. If a communications line is down for several minutes or several hours, this does not stop other parts of the system from performing their tasks. Thus, more work gets done and there are fewer frustrations. TRANSFER handles communications between applications, thus freeing application programs from concern about most exception conditions. TRANSFER assumes responsibility for trying again and again to get data packages to remote nodes that are temporarily inaccessible.

Designers of information systems should implement the concept of loose coupling, where it applies, even if there are no immediate plans to distribute the system. The flexibility inherent in such an approach is worth a great deal. A complex, monolithic application in which all operations associated with particular transactions are tightly coupled is extremely difficult to modify if and when it must be distributed. Thus, TRANSFER should be of interest to designers even when all processing initially takes place on a single node because using TRANSFER makes it easy to distribute the operations later. The location and number of destinations for a package delivered by TRANSFER can be changed just by changing the contents of a distribution list; no re-programming is required.

## Gateways to Other Devices and Computers

In addition to supporting loosely-coupled distributed applications, the TRANSFER delivery system has been designed to support another type of application, sometimes referred to as a *gateway*. Gateways include electronic mail and those applications that manage data flow over communications lines to and from Tandem devices and other devices connected to the Tandem system. (These devices may include facsimile machines, terminals other than the Tandem 6520 and 6530, and other computers.) Making such connections poses some unique problems:

- Most users connecting in this way do not have their own IDs for the GUARDIAN operating system. They connect to the Tandem system via a process running under a GUARDIAN ID, and assume the ID of that process. Clearly some other identity system is required: one that allows users to identify themselves individually with forms of their own names and one in which the number of IDs on a system is unlimited.

- Individual members of this gateway community are normally connected to the Tandem system only for short periods of time. They need an entity that is responsible for processing data on their behalf while they are not in contact with the system.

The TRANSFER delivery system solves the above problems. It defines a 32-character *correspondent* name that can be abbreviated. The maximum number of these names is limited only by the maximum file size on the Tandem system. For each correspondent, TRANSFER defines a *depot,* or private storage areas for a user's data and information. Specific attributes of the depot are as follows:

- Along with a correspondent name, the user must enter the password that is associated with that name to gain access to a depot.

- A *profile* of configuration parameters for each user's environment is maintained in the user's depot.

- *Folders*, or private locations in which to keep information, are a part of the depot. Users are free to add and delete folders within their own depots.

- Users can maintain *distribution lists* in their depots. These list the correspondent names of other users to whom information is to be delivered. Users are responsible for maintaining their own distribution lists, but any user of the TRANSFER system can reference another's distribution lists with read-only access.

## More Features for Loosely-Coupled Applications and Gateways

The TRANSFER delivery system also has the following features for both gateways and loosely-coupled distributed applications:

1. A *package* of data given to TRANSFER is delivered exactly once. TRANSFER assures this by protecting its data base with TMF and relying on TMF recovery to reconstruct the data base in the event of a catastrophe. TRANSFER keeps status information in the package that indicates to whom the package has been delivered. It retains this information until the expiration time has been reached.
2. Any time a package must be shipped to other nodes, TRANSFER transports it in an organized fashion, to one node at a time. It does not need to employ network TMF transactions. Instead, it relies on redundancy in its data base structure to restart *transport* where it left off if the link to the appropriate node goes down during transmission.
3. Once a package has been delivered, the receiving application often needs to act on it. Accordingly, TRANSFER provides for *agents,* user-supplied programs that are selectively invoked upon delivery of a package. The selectivity is uniquely defined for each depot.
4. The standard interface into the TRANSFER system is by message request and reply to a TRANSFER server. Thus, TRANSFER services are accessible to any language that can perform a WRITEREAD, including COBOL, FORTRAN, TAL, and Screen COBOL.

## How the TRANSFER System Works

The TRANSFER delivery system was designed to deliver packages of data reliably anywhere in a network without regard to the format of the data or the availability of resources at any particular moment.[1] In addition, it allows the location of correspondents (either individuals or applications) to change while keeping the linkage to them intact (through distribution lists that reference them). Once TRANSFER has linked two or more applications together, changes in the number or names of recipients does not require program modification.

[1]In the following description of the TRANSFER system, a number of specialized terms are used. These terms are defined in the glossary accompanying this article.

Figure 2.

*The components of the TRANSFER delivery system. TRANSFER can operate on a single Tandem system or on many nodes in a network of Tandem systems. It also provides gateways into the system for other devices and computers.*

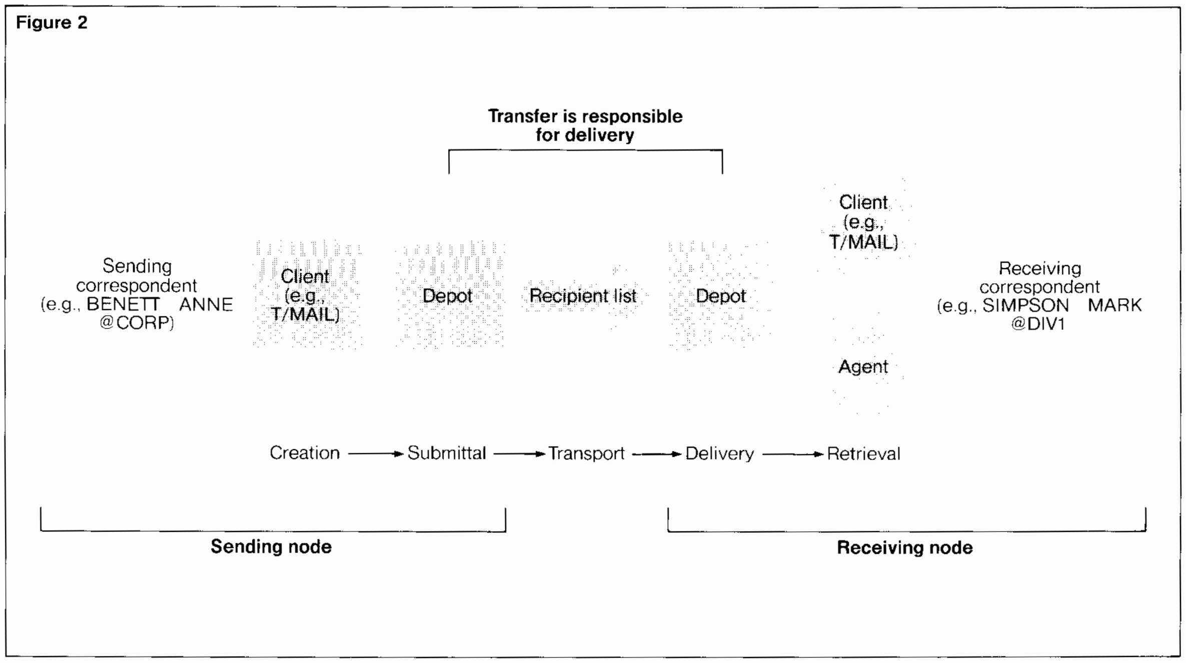


Figure 2
Transfer is responsible for delivery
Sending correspondent (e.g., BENETT ANNE @CORP)
Client (e.g., T/MAIL)
Depot
Recipient list
Depot
Client (e.g., T/MAIL)
Agent
Receiving correspondent (e.g., SIMPSON MARK @DIV1)
Creation → Submittal → Transport → Delivery → Retrieval
Sending node
Receiving node

## Components of the TRANSFER System

Figure 2 depicts the components of the TRANSFER delivery system. On both the sending and receiving end, there are correspondents. These can be people, programs, or devices. A correspondent is assigned a name that specifies the node at which the correspondent receives packages and identifies the correspondent as a unique entity on that node. For example, a correspondent named Anne Benett using the \CORP system would be defined as BENETT_ANNE @CORP.

When correspondents are defined, unique depots are established for them in the TRANSFER data base. (There is no *depot file* per se; *depot* is an abstraction referring to any records that define or pertain to a particular correspondent.) The depot contains a correspondent's distribution lists, profile, and folders. A distribution list is a list of correspondent names and/or the names of other distribution lists to which packages are to be sent. Correspondents can define their own distribution lists. A correspondent's profile contains default delivery parameters, a password, and other necessary information about the correspondent. Folders are filing areas for data that is stored in a depot.

Correspondents communicate their requests to the system via user-interface programs called *clients,* which, in turn, translate the requests for TRANSFER. A client physically links correspondents with their depots. Notice that a client exists at both the sending and receiving end of a delivery. At the sending end, a client creates and submits the package; at the receiving end, another client retrieves it at the correspondent's convenience.

An example of a client is TRANSFER/MAIL (T/MAIL), an electronic mail system supplied with the TRANSFER delivery system. With T/MAIL, correspondents can create mail messages, send them to one or more *recipients*, and read, reply to, file, print, and forward the messages they receive. These correspondent requests are translated by T/MAIL for TRANSFER, which then delivers the mail.

Agent programs resemble clients in that they are user-written, but they act automatically on behalf of a correspondent without the correspondent's having to invoke them, while clients are invoked by the correspondent to interact with TRANSFER.

Clients and agents can be written for any application that uses the TRANSFER delivery system. In the business example in Figure 1, a user-written client would accept the order from a correspondent and send the required information, via requests to TRANSFER, to the corporate office for the credit check. A user-written agent at the receiving end would be notified when the information arrived and would perform as much of an automatic credit check as it could. A user-written client at the receiving end would allow credit personnel to access credit information at their convenience and complete any of the credit check the agent was unable to perform.

# TRANSFER Delivery System Glossary

## Basics

| | |
|---|---|
| *Correspondent* | A user defined in the TRANSFER name data base. |
| *Depot* | A correspondent's private storage areas. |
| *Client* | A program acting as an interface to the TRANSFER system that allows a user to access his depot, e.g. TRANSFER/MAIL. |
| *Agent* | A program invoked automatically during delivery that acts on behalf of a user. |

## Components of packages and depots

| | |
|---|---|
| *Item* | A unit of data created by TRANSFER at the user's request and given a unique ID to allow TRANSFER to keep track of it. |
| *Package* | An item that has delivery information and a list of recipients attached. All packages are items; not all items are packages. |
| *Recipient* | The name of a correspondent or distribution list to which a package is to be sent. |
| *Distribution list* | A list of correspondent names and/or names of other distribution lists. |
| *Folder* | A place within a depot in which to store items. |
| *Inbox* | A special folder in which all packages for a depot are placed upon delivery. Every depot has an inbox. |

## Names and IDs

| | |
|---|---|
| *Session* | A period of interaction with TRANSFER (including access to a depot) which is established by a client when a correspondent name and associated password are presented to TRANSFER. |
| *Session ID* | An internal number that identifies a session to TRANSFER. |
| *Item ID* | An identification assigned to an item that is unique network-wide. It contains node identification and a number unique on the originating node. |
| *Correspondent name* | A 32-character name unique on its node. When combined with the node name, it is unique throughout the network. |
| *Node name* | The 8-character name of a system. |

*(Continued, next column)*

## Phases in the life of a package

| | |
|---|---|
| *Creation* | The generation of items or packages by passing data to TRANSFER. (Performed by a client.) |
| *Submittal* | The giving of a package to TRANSFER for asynchronous delivery. The client signals SUBMIT, and TRANSFER performs the operation. During this phase, each distribution list is expanded and the correspondent names it references are added to the list of recipients. Data within a submitted package is no longer modifiable. |
| *Delivery* | The placing of a pointer (the item ID) in the inbox folder of every recipient on the local node that is to receive a specific package. (Performed by TRANSFER.) |
| *Transport* | The transfer of a package to each remote node in a list of recipients, one at a time. When the TRANSFER system on the remote node receives the package, it assumes responsibility for processing the package on that node. |
| *Retrieval* | The perusal of a correspondent's inbox at the correspondent's convenience. (Performed by a client.) |

## Time stamps on a package

| | |
|---|---|
| *Delivery begin* | A time specified in a package before which the package must not be delivered. |
| *Delivery end* | A time specified in a package after which the package must not be delivered. If it is still undelivered when this time is up, the package is returned to the sender. |
| *Expiration of a package* | A time specified in a package after which, if the package is still unexamined, it is returned to the sender. |

## Phases in the Life of a Package

There are five phases in the life of a package:

- *Creation.* The package is introduced into the system.
- *Submittal.* The package is committed to be sent to one or more destinations.
- *Delivery.* The package arrives at local destinations on time and complete.
- *Transport.* The package is transferred to one or more remote nodes.
- *Retrieval.* The package is read and processed.

The following describes what takes place during each phase.

***Creation.*** When a client creates a package of information, that package consists of *items* and recipient names. The items are composed of records in the TRANSFER data base containing the data to be sent. A package can consist of one or more items and have a list of one or more recipients. A recipient can be an individual correspondent or a distribution list.

*TRANSFER checks the availability of the remote node regularly.*

Package creation occurs when the TRANSFER system creates an item of type PACKAGE. Individual recipient name and data records, containing the item ID returned by TRANSFER, are then added. Other whole items may also be attached to the package, and it may continue to be added to or modified until it is submitted.

***Submittal.*** When the client submits the package, TRANSFER assumes the responsibility for delivering it to all the recipients. The TRANSFER system does this asynchronously, freeing the client to process other requests from the user.

TRANSFER first expands the recipient list for the package. If the list contains any local distribution lists, TRANSFER adds all correspondents on these distribution lists to the recipient list. (If any duplicate correspondent names are found, they are discarded so that the package is delivered only once to each correspondent.) Thus, associated with each package is an original recipient list and, possibly, an expanded list.

***Delivery.*** Once the recipient list has been resolved, the TRANSFER system delivers the package to local recipients. If the package is submitted as a timed delivery (i.e., it is not to be delivered until a specified time), TRANSFER delays the local delivery until the time specified in the package.

TRANSFER delivers a package by depositing it in the *inbox* folder in the recipient's depot. (The TRANSFER system supplies all depots with an inbox folder. A client can create other folders, if desired, and use them to store related packages).

Once the package has been inserted in the inbox folder, the TRANSFER system checks to see if an agent should be invoked upon receipt of the package. Each depot can be defined (through the T/MAIL administration client or a user-written client) as having one or more agents invoked upon delivery. If agents are defined for the type of package that is delivered, TRANSFER invokes them one at a time in the sequence defined in the depot. After all agents have been invoked, TRANSFER continues to deliver the package to other recipients.

After the local delivery has been completed, TRANSFER checks the package to see if the client specified that the package is to have an expiration date. If so, TRANSFER inserts the package in a time queue.

The TRANSFER delivery system assures delivery of the information within a specific period of time (measured in minutes or hours, not seconds). If the package cannot be delivered within the specified period, TRANSFER returns the package to the sender, enclosed within an error package. (For example, a package might be returned to the sender because the path to the appropriate node was unavailable within the specified delivery period or the package was delivered but not retrieved by the recipient before it expired.)

***Transport.*** If any of the recipients resides on a remote node and a path to that remote node is available, the TRANSFER system transports the package. If a path is not available, TRANSFER saves the request and transports the package when it is available.

TRANSFER checks the availability of the remote node regularly. As soon as it has determined that a path to the node is available, it transports the package.

The TRANSFER system transports a package in three steps. First, it copies and inserts the items in the package into the TRANSFER data base on the destination node. (All items have network-wide unique identifiers. If an item already exists on the remote node, it is not copied; instead, TRANSFER simply points to the existing copy.) Next, it transmits the original recipient list and the recipients for the remote node to the remote data base. Finally, it formats a submit request to the TRANSFER software running on the remote node, which then assumes responsibility for recipient list expansion, delivery, and transport to other nodes referenced in the expansion of the recipient list. The local TRANSFER system is left free to continue local processing.

***Retrieval.*** At their convenience, recipients run a client program to retrieve packages in their depots. For example, they can run the client T/MAIL, supplied by Tandem, to retrieve mail messages when they are ready to read them. Application programmers can write other clients to perform the retrieval and processing functions necessary for their applications.

### Asynchronous Processes

The interface to TRANSFER is generally performed by an interactive client running at a priority consistent with interactive applications. The following portions of TRANSFER are designed to operate in the background asynchronously:

- Transport between nodes.
- Timed delivery at a node.
- Execution of agents triggered by delivery.

This asynchronous part of TRANSFER is configured to run at a lower priority than that of the interactive interface, so that it operates only when the interactive environment is not consuming all the CPU power. That is, it is designed to make use of the excess capacity available between the peaks in utilization created by interactive applications.

## Conclusion

Distributed applications and gateways have unique information-delivery requirements. While the delivery needs of individual OLTP applications are easily met with the ENCOMPASS distributed data base system, loosely-coupled applications and gateways also need the delivery service of the TRANSFER system.

The TRANSFER system provides the following features:

- Assured delivery of information. If the information cannot be delivered within the specified period of time, TRANSFER automatically notifies the sender.
- Elimination of duplication on a node. If an item is already on a receiving node, it is not transported again. Also, a package is delivered to each recipient only once, even if the recipient is in more than one distribution list referenced by the package.
- Transport across the EXPAND network.
- Timed delivery of packages on a node.
- Automatic invoking of user programs when a package is delivered.
- A programmatic interface between user-written programs and the TRANSFER software.

The TRANSFER delivery system is flexible, reliable, and data-format-independent. TRANSFER handles the transport and delivery of data on behalf of users, without requiring them to be logged on to the system at that time. It is a general-purpose delivery system that greatly increases the speed and efficiency with which loosely-coupled distributed applications and gateways can be implemented. It also allows for the easy expansion and modification of the applications as they grow. Finally, the TRANSFER delivery system allows the application designer to create a distributed application based on the natural interaction of the company's operating areas without having to implement a complex delivery and transport mechanism.

**Steve Van Pelt** is a senior systems analyst in Tandem's Customer Application Support Group. For the past year and a half he has been primarily involved in training Tandem analysts in the TRANSFER delivery system. Since joining Tandem in January 1981, Steve has also trained Tandem analysts in relational data base design and the PATHWAY transaction processing system. He has a Bachelors of Science degree in Electrical Engineering from Leland Stanford Junior University.

# An Introduction to Tandem EXTENDED BASIC

Tandem EXTENDED BASIC is a simple, interactive programming language that is particularly useful for quick problem-solving applications. It meets the 1978 ANSI X3.6 standard for BASIC, as well as offering many useful extensions. It comes with both an interpreter and a compiler. When the interpreter is used, each source line is checked for errors as it is typed in by the user, and all correct input is immediately processed. The compiler can be used to change user-supplied source lines into object code, providing for faster execution. This article introduces the features of Tandem EXTENDED BASIC.

## Accessing EXTENDED BASIC

On most systems having EXTENDED BASIC, the programmer can run the interpreter by simply typing the Command Interpreter command BASIC. The compiler is invoked from within the interpreter environment.

Interpreter commands enable the BASIC programmer to create, edit, compile, run, and debug programs. They perform functions outside the context of the program and are not a part of the BASIC program itself. Unlike BASIC statements, which must have line numbers, interpreter commands are entered without line numbers and execute immediately. EXTENDED BASIC provides an arrow (→) prompt when ready for its next command.

**Table 1.**
A summary of commonly used interpreter commands.

| Command | Function |
|---|---|
| APPEND | Merges another source program with the current program. |
| BINSAVE | Saves pseudo-code to a permanent file. |
| COMPILE | Produces object code for the current program. |
| CONT | Continues interpretive program execution after a STOP statement. |
| DELETE | Removes line(s) of the current program. |
| EXIT (CTRL/Y) | Exits the BASIC environment. |
| F | Enables editing of the current source line (similar to the EDIT FIX function). |
| FC | Enables editing of last interpreter command entered (similar to the Command Interpreter FC command). |
| FILES | Lists all files in a particular subvolume. |
| HELP | Prints syntax for BASIC interpreter commands and BASIC statements. The default is to print all available classes of information in the HELP file. |
| LENGTH | Prints the amount of memory used by the current program. |
| LIST | Prints the specified line(s) of the current program. |
| NEW | Clears the interpreter memory area in order to create a new program. |
| OLD | Loads a specified program (either source or pseudo-code) into the interpreter's memory area. This program becomes the current program. |
| RENAME | Changes the name of the current program. |
| RUN | Executes the specified program (current program by default). |
| SAVE | Saves the source text of the current program to a permanent file. SAVE ! saves the text in the same file; otherwise, a filename must be specified with this command. UNSAVE purges a specified file. |
| TRACE | Causes the printing of line numbers during program execution. NOTRACE disables this facility. |
| VOLUME | Resets or displays the default volume and subvolume. |

A summary of the more commonly used interpreter commands is in Table 1.[1]

When EXTENDED BASIC loads a source program, it translates it into an executable form, called pseudo-object code. (Pseudo-object code is not humanly readable, but is understandable to the interpreter.) The process can be quite time consuming, so EXTENDED BASIC provides a way to save the pseudo code in a file, and then run the pseudo code without having to reinterpret a program's source.

It may be desirable to generate object code from the EXTENDED BASIC source since object code executes much faster than pseudo code. This can be done by issuing the interpreter command COMPILE. Often during a program's development, the source code is written, run, and debugged through the interpreter, and then a "final" bug-free copy of the program is compiled into an object program that is released to the user community.

When the interpreter is used, any changes made to the current program are not reflected on the permanent file copy of the program unless a SAVE command is issued. The programmer must be careful to SAVE altered programs, as it is entirely possible to make a large number of changes to a program through the BASIC interpreter, exit the interpreter without issuing a SAVE command, and lose those changes.

EXTENDED BASIC variable names can be up to 29 characters in length. The first character must be alphabetic, but the remaining characters may be alphabetic, numeric, or a period (.). The last character of the name determines the variable type:

| **Last character in name** | **Resulting variable type** |
|---|---|
| % | integer |
| $ | character |
| any other | default numeric (either FIXED or REAL) |

The default numeric type is either FIXED decimal point representation with nine fractional digits, or, if the Floating Point Option is used, 64-bit exponential (REAL) representation.

Line numbers must be given for every BASIC line entered. Source lines are sorted in ascending order based on the line number. To define more than one statement in a single program line, statements can be separated by back slashes (\), in which case the line number labels the first statement of the line. EXTENDED BASIC also provides a facility to continue a program line onto the next input record by entering an ampersand (&) as the last character on the line to be continued.

## EXTENDED BASIC Statements

EXTENDED BASIC statements fall into two categories: executable and non-executable. Executable statements specify "actions;" they control the input and output of data, evaluate mathematical calculation, and control program flow. The more commonly used executable statements and their operation are summarized in Table 2.

*EXTENDED BASIC has both an interpreter and a compiler.*

Non-executable statements are declarative statements used to specify data items and functions used in a program. Also included is the REM statement which allows comments to be placed in the middle of program code. The more commonly used non-executable EXTENDED BASIC statements are summarized in Table 3.

## Program Flow

The flow of an EXTENDED BASIC program is statement-by-statement, starting at the top of the program, and proceeding sequentially downward. This flow can be interrupted by transfer statements, looping, subroutine invocation, and error recovery.

[1]In the tables in this article, *current program* refers to the program currently stored in the interpreter memory area (i.e., the one last referenced in an OLD or NEW interpreter command).

**Figure 1.**

*Syntax for the IF statement, one of the fundamental executable statements in BASIC.*

**Figure 1**

```
IF ‹condition› {THEN ‹statement›   } [ELSE ‹statement›   ]
               {THEN ‹line-number›} [ELSE ‹line-number›]
               {GOTO ‹line-number›}
```

Transfer statements cause the control of execution to be set to a different statement, whereupon the sequential program flow is resumed. It may skip over many statements, or jump backward to re-execute statements already encountered. A GOTO statement, like the one in the example below, is a transfer statement.

```
10  GOTO 40     ! Execute 10, transfer to 40
20  i% = 5      ! Will not be executed
30  j% = 6      ! Will not be executed
40  k% = 7      ! Will be executed, natural
                !     sequence resumed
```

Note that program execution transfers immediately and unconditionally to the specified line number. (In this and all successive examples, EXTENDED BASIC reserved words are capitalized and user-supplied names are shown in lower case.)

IF-THEN and IF-GOTO statements conditionally transfer program execution to a specified line number, depending on the outcome of a test. The syntax of the IF statement is shown in Figure 1.[2]

The deciding ‹condition› is a relational expression which must evaluate to a TRUE or FALSE value. If the condition is true, the first (or TRUE) clause is executed. If the condition is false, then one of two actions may

[2]In all syntax diagrams, the notation of the *EXTENDED BASIC Reference Manual* is used: braces ({}) indicate that exactly one of the options listed must be selected; brackets ([ ]) indicate that the field is optional and any number, including zero, of the enclosed options may be chosen; angle brackets (‹›) indicate a field in which the user must supply the variable name or expression.

**Table 2.**
A summary of commonly used executable statements.

| Statement | Function |
|---|---|
| CHAIN | Transfers control from the current program to another program and commences execution of the new program. |
| CLOSE | Removes (from the program) access to a file until it is reopened. |
| DELETE | Removes an existing record from a file. |
| FIND | For use with record-oriented file access, sets the current and next record pointers to a specified record. |
| FOR, UNTIL, WHILE | Allows repeated execution of a set of statements. A FOR loop executes for as many times as the programmer specifies; an UNTIL loop executes while the control expression is FALSE; a WHILE loop executes while the control expression is TRUE. |
| GET | Reads a record (pointed to by the current record pointer, or specified by a key value) from a file into a program variable or variables. GET is the recommended way to receive non-terminal input. |
| GOSUB | Transfers control of execution to a designated line where execution continues until a RETURN statement is encountered. |
| GOTO | Transfers control of execution to a designated line. |
| IF | Transfers the program execution point to different places, depending upon whether a specified expression evaluates to a TRUE (nonzero) or FALSE (zero) value. |
| INPUT, LINPUT | Allows user entry of data (usually from a terminal) during program execution, with the optional display of a prompt to the user. INPUT is oriented toward reading a series of data items, while LINPUT is oriented toward reading an entire line as a single data item. |
| KILL | Purges a file. |
| MAP | Allocates a section of memory as a buffer and associates a set of variables with it. |
| MAT | Performs matrix operations on arrays. The operations include array addition, subtraction, or multiplication; initialization to zero, one, or the identity matrix; transposition; and inversion. |
| NEXT | Used with FOR, WHILE, and UNTIL statements to mark the end of a loop of statements. |
| ON ERROR GOTO | Provides a method for a program to trap execution errors and perform user-defined error recovery. |
| OPEN | Readies a file for data transfer between a physical file (identified by a logical channel) and program variables. |
| PRINT | Writes data (as ASCII characters) to a specified file (usually the user terminal). |
| PRINT USING | Writes formatted data to a specified file (usually a terminal). Character masks are used to determine the way in which the output of strings or numerics is to appear. |
| PUT | Writes a record to a file. This is the most effective way to send information to a non-terminal physical device. |
| READ | Inputs a list of values built into a data pool by one or more DATA statements. |
| RESUME | Allowed only in error trap routines, resets error flag and begins program execution at the specified line. |
| RETURN | Transfers execution point from within a subroutine back to the place where the subroutine was invoked (the next line after the GOSUB statement). |
| SORT | Reorders a set of records according to a specified key field. |
| STOP | Suspends program execution. |
| UPDATE | Replaces or deletes an existing record (pointed to by the current record pointer). |
| WAIT | Designates the maximum amount of time the system will wait for I/O operations (e.g., user terminal input) to complete. |

occur. If a second (or FALSE) clause is given, it is executed. If no ELSE clause has been specified, program execution resumes at the next sequentially numbered line after the IF statement.

Below are two example IF statements. In the first one, if c% is equal to 0, answer$ is set to "zero"; otherwise, it is set to "non-zero." In the second, the absolute value of x is calculated.

```
430  IF  c% = 0 THEN answer$ = "zero"   &
                  ELSE answer$ = "non-zero"

610  IF x < 0  THEN x = -x
```

The ON GOTO construct can be used to transfer execution control to one of several statements. A numeric expression is evaluated (usually to a very small positive integer) and then used as an index to a list of statements to which to transfer. If the value of the expression is equal to one, then program control passes to the first line specified; if the value is two, control passes to the second line; and so on. An example better illustrates this:

```
40  ON b+3  GOTO 10, 20, 300
```

If b has a value of −2, transfer is made to line 10; if b has a value of −1, transfer is made to line 20; and if b has a value of 0, transfer is made to line 300. An error is generated if the numeric expression (in this case, b+3) evaluates to an index without a corresponding line number.

A program loop is a series of statements written so that control transfers to the first statement of the series after the final statement of the loop is executed. The process continues until some terminating condition is reached. There are four types of loop constructs within EXTENDED BASIC:

- FOR/NEXT loop
- UNTIL/NEXT loop
- WHILE/NEXT loop
- Creation of a loop through the use of a GOTO statement

A loop causes the same set of statements to be repetitively executed, either a fixed number of times, or until a condition is (or is no longer) met. Each of these constructs is described in the next section.

Subroutines are invoked via the GOSUB verb. Unlike other high-level languages, BASIC does not require that a subroutine correspond to physical blocks of code segregated from the rest of the program. A subroutine is defined as those statements executed between the time a GOSUB statement and a RETURN statement are encountered. The use of subroutines is explained in more detail in the section, "Invoking a Subroutine."

Error recovery can be handled by the system if the programmer chooses not to intercede. The default EXTENDED BASIC action is to terminate execution and display an error message corresponding to the error. EXTENDED BASIC provides the means to trap execution errors and allow user-defined error recovery through the ON ERROR GOTO construct. Control is then passed to a specified line number once an error occurs, where the error can be determined and appropriate action taken. Error recovery is discussed in greater detail in the section, "Error Handling."

**Table 3.**
A summary of commonly used non-executable statements.

| Statement | Function |
|---|---|
| DATA | Introduces a constant value, or series of constants, into a program. Data appearing in a DATA statement is read into the program by the use of one or more READ statements. |
| DECLARE | Defines the type of a numeric variable, array, or function. EXTENDED BASIC types are 16-, 32-, and 64-bit integer; fixed-point; and 32- or 64- bit floating-point real representations. |
| DEF, FNEND | Defines a sequence of operations that return a single numeric or string value as a user-defined function. DEF statement is used to specify the function name and returned value. FNEND indicates the end of the text if it is a multi-line function. |
| DIM | Used to specify the maximum number of elements in an array. |
| IMAGE or : | IMAGE, or a colon (:), is used to give the format description for values being output by a PRINT USING statement. |
| MAP | Allocates a section of memory as a buffer and associates a set of variables with the space. |
| REM or ! | Used for placing programmer comments in the program code, in order to better document the program. Text appearing in a REM statement, or to the right of an exclamation point (!) is ignored by the interpreter. |

## Loop Constructs

A loop is a self-repeating sequence of program statements. Loop constructs can be built in four different ways. The most explicit method, and probably the best for most applications, is the use of a FOR/NEXT loop. Here the programmer states the exact number of times a loop is to be executed. The WHILE and UNTIL loop constructs are very similar to each other; each is dependent on a stated condition being TRUE or FALSE to determine how many times a loop is to be executed. The final (and least structured) way to build a loop is by the use of a GOTO statement that directs execution to a point in the program preceding the GOTO statement.

The FOR loop has the following syntax:

```
FOR ‹loop index› = ‹lower bound› TO          &
          ‹upper bound› [STEP ‹increment›]
...
NEXT ‹loop index›
```

If ‹lower bound› is greater than ‹upper bound› before the loop is entered, the loop is not executed. Otherwise, the loop is executed with ‹loop index› having a value of ‹lower bound› the first time through the loop. ‹Lower bound› is then incremented by ‹increment› each time through the loop, and the loop is repeated until the value for ‹loop index› exceeds that of ‹upper bound›. The value for ‹loop index› can be used within the loop for processing. The default ‹increment› is one.

Figure 2 illustrates how a FOR/NEXT loop is constructed. The example prints the square of the integer values between 1 and 10.

Any number of loops can be nested as long as all inner loops are fully imbedded within the next most inner loop. The following is a program segment that initializes a 3 x 4 array to values of zero using an imbedded loop (lines 60 through 80):

```
50  FOR x = 1 TO 3
60     FOR y = 1 TO 4
70        array (x,y) = 0
80     NEXT y
90  NEXT x
```

Figure 3 illustrates correct and incorrect loop nesting.

The UNTIL and WHILE clauses allow the creation of conditional program loops. The UNTIL statement has the following syntax:

```
UNTIL ‹numeric expression›
  ...
NEXT
```

The syntax for the WHILE statement is identical:

```
WHILE ‹numeric expression›
  ...
NEXT
```

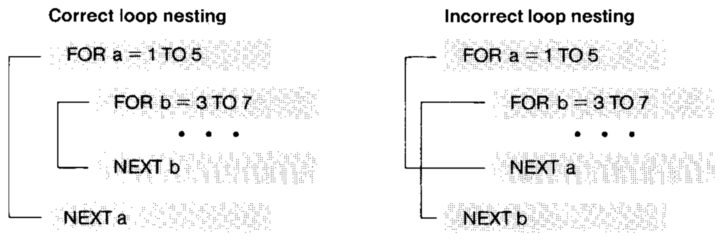
Figure 2

FOR/NEXT loop

```
10  FOR i% = 1 TO 10
20        square% = i% * i%
30        PRINT i%, square %
40  NEXT i%
50  END
```

**Figure 2.**
*In this example of a FOR/NEXT loop, the value of the loop index (i%) and its square will be printed for the integer values 1 through 10. A FOR/NEXT loop is executed a fixed number of times (in this instance, 10 times).*

Figure 3


Correct loop nesting

```
┌─ FOR a = 1 TO 5
│  ┌─ FOR b = 3 TO 7
│  │     • • •
│  └─ NEXT b
└─ NEXT a
```

Incorrect loop nesting

```
┌─ FOR a = 1 TO 5
│  ┌─ FOR b = 3 TO 7
│  │     • • •
└──┼─ NEXT a
   └─ NEXT b
```

**Figure 3.**
*Correct and incorrect loop nesting. In the correct example on the left, b will be incremented from 3 to 7 before a is incremented, since a is in an inner loop.*

‹Numeric expression› is evaluated at the beginning of each iteration of the loop. The statements of the UNTIL loop are executed until ‹numeric expression› becomes TRUE, at which point control is transferred to the statement immediately following the NEXT statement. WHILE loops are the corollary to UNTIL loops: a WHILE loop is executed until ‹numeric expression› becomes FALSE.

The FOR/NEXT loop example of Figure 2 can be coded using UNTIL or WHILE loops as shown in Figure 4. When an UNTIL loop or a WHILE loop is used, care should be taken to insure that the loop can be exited. The loop condition clause should change in value for subsequent times through the loop such that a FALSE situation for a WHILE loop (TRUE for an UNTIL loop) can be reached. Alternatively, the logic within the loop should contain a transfer statement (i.e., GOTO) such that the loop can be exited. There are only rare occasions (such as reading to the end of a file) when it is desirable to code an infinite loop.

The final way to create a loop is through the use of a "backward-referencing" GOTO statement. In backward-referencing, the target statement of the GOTO appears before the GOTO statement. This method is cruder than the other loop constructs and generally should not be used. Squaring ten integers would be coded as shown in Figure 5.

**Figure 4**

**UNTIL/NEXT loop**
```
10  i% = 1
20  UNTIL i% > 10
30      square% = i% * i%
40      PRINT i%, square%
50      i% = i% + 1
60  NEXT
70  END
```

**WHILE/NEXT loop**
```
10  i% = 1
20  WHILE i% <= 10
30      square% = i% * i%
40      PRINT i%, square%
50      i% = i% + 1
60  NEXT
70  END
```

**Figure 4.**
*Examples of WHILE/NEXT and UNTIL/NEXT loops. These examples print the value of the squares of numbers between 1 and 10.*

**Figure 5**

**Backward-referencing GOTO statement**
```
10  i% = 1
20  square% = i% * i%
30  PRINT i%, square%
40  i% = i% + 1
50  IF i% <= 10 GOTO 20
```

**Figure 5.**
*Example of a backward-referencing GOTO statement to create a loop. Until the value of i% exceeds 10, the GOTO at line 50 will force execution to continue at line 20. This program also prints the value of i% and its square for values between 1 and 10.*

## Terminal I/O

Two special EXTENDED BASIC statements handle communication to and from the terminal. The INPUT statement can be used to provide data to a program while it is running. It has this syntax:

```
INPUT [" ‹prompt› " ,] ‹variable list›
      [" ‹prompt› " ;]
```

‹Prompt› is a character string that is displayed on the user's terminal, before that user can respond to the program with an answer. ‹Variable list› can contain any number of numeric or string variables in any order, for which the user must supply a corresponding value. When an INPUT statement is encountered, the program waits until enough data has been typed in by the user. EXTENDED BASIC prompts for terminal input with a question mark (?).

The other special statement for terminal I/O is the PRINT statement, which handles output to the terminal. In the simplest case (the word PRINT with no variables supplied), the PRINT statement outputs a blank line. For more sophisticated use, the syntax is:

```
PRINT [‹variable›] [, ‹variable›]...
                   [; ‹variable›]...
```

‹Variable› may contain any expression, character string, numeric constant, or variable. If a comma (,) is used as the separator between variables, spaces are inserted on the output stream to give column alignment. If a semicolon (;) is used as the separator, no spaces are inserted between consecutive output fields.

**Figure 6.**

*Use of the INPUT and PRINT statements. The values in boldface type are user input. The INPUT statement prompts for input and then waits to receive the values from the user.*

**Figure 6**

**Source code**

```
10  REM     Calculate hypotenuse of right triangle.
20  INPUT   "Enter the first side", a
30  INPUT   "Enter the second side", b
40  PRINT   "Hypotenuse:"; sqr((a*a) + (b*b))
50  END
```

**Sample run**

```
Enter the first side    ? 3
Enter the second side ? 4
Hypotenuse: 5
```

**Figure 7.**

*Formatting with the PRINT Statement. Note how the output differs with commas and semicolons. (This example contains a complete interpreter session.)*

**Figure 7**

**Source code**

```
10  dim x$(5)
20  x$(1) = "a"
30  x$(2) = "   b   "
40  x$(3) = "   c   "
50  x$(4) = "   d   "
60  x$(5) = "   e   "
70 print x$(1), x$(2), x$(3), x$(4), x$(5)
80 print "Now with semicolons..."
90 print x$(1); x$(2); x$(3); x$(4); x$(5)
100 end
```

**Sample run**

```
:basic
EXTENDED BASIC FX - T9204A00 - (01AUG83)
→old example
→run
$TRAIN.JMBASIC.EXAMPLE   01-Feb-84   17:45:29
a              b         c          d          e
Now with semicolons...
a   b     c     d     e
CPU time : .07 seconds
→EOF!
Total CPU time : .48 seconds
Elapsed time : 1 minute 2.44 seconds
Thank You.
:
```

The program in Figure 6 illustrates the use of the INPUT and PRINT statements. Note the semicolon in the PRINT statement; it insures that the two print fields are not separated by extra white space. Figure 7 represents another program to show how output formatted with commas and semicolons can differ.

## Program Data

Like the INPUT statement, the READ and DATA statements supply data to the program. However, in this case, the data is contained in the program itself, rather than coming from an external source (terminal or file). The READ and DATA statements differ from INPUT in another way. When INPUT is used successive times, the data that is received by the program can differ from one occurence to another. READ and DATA supply a fixed list of data values to the program, and the program must be edited to change the values.

A READ statement inputs the list of variables whose value it obtains from a DATA statement. Neither statement works without the other. A READ statement has the syntax:

READ ‹variable› [,‹variable›]...

‹Variable› may be a numeric variable, string variable, or an array.

The DATA statement consists of a series of constants, as follows:

DATA ‹constant› [,‹constant›]...

‹Constant› may be either a string or numeric constant. With a READ statement, the variables listed are assigned values sequentially from the set of DATA statements in the program. Before the EXTENDED BASIC program is actually run, all DATA statements are appended to each other in a data block. Each time a READ is encountered, the next value from the data block is extracted and placed in the variable mentioned in the READ statement. An error occurs when a READ statement is encountered and the data in the data block has been exhausted.

The following example shows the use of these statements:

```
150  READ x, y%, z$
350  DATA 6, 20, 30, 50, 70, 85
```

After this fragment is executed, x will have the numeric value 6.0, y% will have the numeric value 20, and z$ will have the character value "30." The next variable in a subsequent READ statement (not shown in this fragment) would be assigned the constant 50.

Often, it is necessary to use the same DATA data more than once in a program. The RESTORE statement causes the next READ statement to begin reading data from the first DATA statement in the program, regardless of where it read the last data value.

## File I/O

Among the most useful features of EXTENDED BASIC are those having to do with file access. A file can be a mass storage file, a tape file, a line printer device, or another process. The user can store information read in or calculated during one program session and then retrieve and possibly update the information at a later time. Tandem EXTENDED BASIC offers seven file types, summarized below:

*Unstructured files.* Unstructured files have data stored as a string of bytes. Data is written to the file as records of any length (up to the record length specified when the file was opened). Records are always padded to an even number of characters. Data may be read sequentially or by explicit record number. Any file (even if created as another file type) can be accessed as an unstructured file.

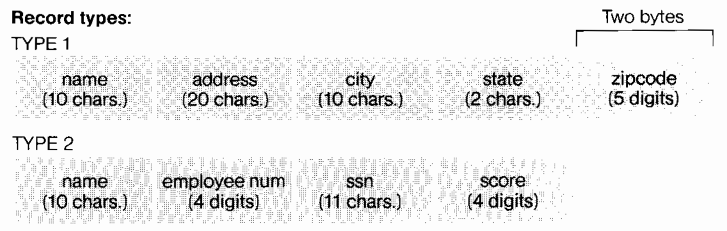
*Fixed files.* Fixed files are unstructured files, containing records of fixed length. Data is written to the file in records of the record length. (Shorter records are padded with blanks.) Data may be accessed sequentially or by record number.

*Edit files.* Edit files are a special type of unstructured file. Each record is subjected to a compression algorithm, may be up to 256 characters in length, and has an associated EDIT line number. At any one time, an EDIT file cannot be opened for both READ and WRITE operations. Data may only be accessed sequentially. This file type is used by the Tandem text editor, EDIT.

*Virtual files.* In EXTENDED BASIC, virtual files exist to store the information contained in virtual arrays. (Virtual arrays exist to handle extremely large blocks of information—up to 2M bytes—that can be referenced by a single variable name.) The only permissible way to access information from a virtual file is by reference to array elements.

*Indexed files.* Indexed files consist of variable-length records, each record containing a unique primary-key field. Records are arranged in ascending order of primary-key field values. Record lengths can vary from one byte to the maximum specified when the file is created. The records can be accessed either sequentially or randomly through primary or alternate keys.

**Figure 8**



**Format for reading them into a file:**

```
MAP(type1) name$= 10, address$= 20, city$= 10, state$=2, zipcode%
MAP(type2) name$= 10, empno%, ssn$= 11, score
```

**Figure 8.**
*Use of the MAP statement. In this example, a format for reading the two record types (type1 and type2) above is defined.*

*Relative files.* Relative files consist of variable-length records stored in a position relative to the beginning of the file. Each record is assigned a record number (based on the record position) that uniquely identifies it. Record lengths may vary in size from one byte to the maximum size specified when the file was created. They can be accessed sequentially or randomly by record number or alternate keys.

*Sequential files.* Sequential files consist of variable-length records stored in the sequence in which they are entered. Alternate keys are supported. Record lengths may vary from zero bytes to the maximum size specified when the file was created. New records are appended at the end of the file.

The default file type when none is specified is an EDIT file. Further information on these file types can be found in the *ENSCRIBE Programming Manual.*

The MAP statement is used to define record layouts to the BASIC program. The syntax for its use is:

```
MAP (<mapname>) { <var1> [= <bytesize>] }
                { FILL                   }
                { FILL%                  }...
                { FILL$ [= <bytesize>]   }
```

The example in Figure 8 better illustrates how the MAP statement is used. For memory storage on a Tandem system (in most languages, not just EXTENDED BASIC), a character takes one byte of storage. In Figure 8, both name$ and city$ are given 10 bytes of mapped storage area; address$ is given 20 bytes.

**Figure 9.**

*Syntax for the OPEN Statement. ‹Filename› represents a file name on the Tandem system. It must be enclosed in double quotation marks (" ") or given in a string variable. The FOR INPUT clause indicates that the file must exist; an error is generated if it does not. The FOR OUTPUT clause purges a file if it exists and creates a new file (of the same name) with the attributes specified. ‹Fileno› must be a number between 1 and 63, and represents the logical channel number associated with the file.*

**Figure 9**

```
OPEN ‹filename› [FOR {INPUT}  ] AS [FILE] [#] ‹fileno›
                     {OUTPUT}

                   {EDIT         }
[,[ORGANIZATION]   {FIXED        }]
                   {INDEXED      }
                   {RELATIVE     }
                   {SEQUENTIAL   }
                   {UNSTRUCTURED }
                   {VIRTUAL      }

          {READ   }          {NONE  }
[,ACCESS  {MODIFY } ] [,ALLOW {READ  }]
          {WRITE  }          {WRITE }
          {APPEND }          {MODIFY}

[, {MAP ‹mapname›          }] [, PRIMKEY ‹keyname›]
   {RECORDSIZE ‹recsize›}
```

**Figure 10**

```
                 [, {RECORD ‹recno›                              }]
GET #‹fileno›    [, {KEY [#‹prim/alt key›] {GT/GE/EQ} ‹string›   }]
                 [, {KEY [‹keyid›]         {GT/GE/EQ} ‹string›   }]

PUT #‹fileno›    [, RECORD ‹recno›] [, COUNT ‹bytecount›]
```

**Figure 10.**

*Syntax for the GET and PUT statements. GET and PUT are the recommended statements for communicating with permanent files and non-terminal devices.*

The variables empno% and zipcode% get the default storage allotment for integer variables (2 bytes) and score the default size for a fixed-point variable (8 bytes).

For a logical record called type3, based on the type2 record in which the social security number (ssn) in Figure 8 is not used, the following record would be designed:

```
MAP(type3) name$= 10, empno%,          &
           FILL$= 11, score
```

Use of the MAP statement is one of the more difficult aspects of EXTENDED BASIC programming; however, the ability to define logical record formats from within the program is powerful and effective. It allows the programmer to access parts of a record directly, instead of having to fill temporary buffers and extract the portion needed. The capability of creating logical overlays in order to represent the record data in more than one way is available in only a few versions of BASIC in addition to Tandem EXTENDED BASIC. MAP statements are often used in conjunction with the OPEN statement to specify the record formats of the file to be OPENed.

The OPEN statement is used for data transfer to and from a file. It sets up the logical channel used to reference the file within the EXTENDED BASIC program and optionally creates the file if it does not already exist. The ACCESS and ALLOW clauses of the OPEN statement are used to control the manner in which a file is used. The syntax for the OPEN statement is shown in Figure 9.

The ORGANIZATION clause states the structure of the file to be opened. An error is generated if the existing file organization does not match the stated organization, unless UNSTRUCTURED is selected, in which case the file is opened as unstructured. If the ORGANIZATION clause is omitted, the file is opened with its existing organization mode. If the file does not exist and no ORGANIZATION clause is given, it is created and opened as an EDIT file type.

The ACCESS clause of the OPEN statement specifies which file operations can be performed by the process that opened the file. The types of access are defined below:

*READ*—Allow read-only access by the user.

*WRITE*—Allow write-only access by the user.

*APPEND*—Position to the end of the file and allow write access by the user.

*MODIFY*—Allow read, write, delete, and update access by the user.

The ALLOW clause of the OPEN statement is used to specify the operations that other processes are permitted to perform on the file after it has been opened in the EXTENDED BASIC process:

*ALLOW NONE*—Allow no other process to use the file while it remains open.

*ALLOW READ*—Allow other processes to read the file while it is open.

*ALLOW WRITE* and *ALLOW MODIFY*—Allow other processes to read, write, and update the file while it is open.

The MAP clause of the OPEN statement matches a MAP buffer (created in a MAP statement occuring before the OPEN statement) with the information to be retrieved from the open file. The MAP clause also defines the record size and keys, when applicable.

RECORDSIZE defines the maximum record size (in bytes) to be read from the opened file. RECORDSIZE and MAP clauses cannot appear in the same OPEN statement.

The primary statements used to communicate with an opened file are GET, which reads a record from the file into a program variable or buffer space, and PUT, which writes a record to the file. The syntax for the two statements is shown in Figure 10.

There are other I/O operations, besides the primary ones covered so far, that can be performed on permanent files. These include resetting the current record pointer (FIND statement), deleting the current record (DELETE statement), updating the current record (UPDATE statement), and closing files (CLOSE statement). An example program using I/O operations is shown in Figure 11.

**Figure 11**

```
10      ON ERROR GOTO 19000
20      MAP(type2) name$=10, empno%, ssn$=11, score
30      OPEN "file1" AS #1, edit, ALLOW none, MAP type2
40      GET #1
50      score = score + 1
60      UPDATE #1
70      GOTO 32767
19000   IF err = 12 THEN PRINT "File Busy."                    &
            \GOTO 32767
19999   ON ERROR GOTO 0       ! Unexpected error
32767   END
```

**Figure 11.**
*An example of updating a record. This program opens a file in exclusive read-and-write-mode, updates a record in the file, and releases the file.*

## Error Handling

EXTENDED BASIC has built-in error handling to improve the structure and readability of program code. It is necessary to "turn on" the error trapping by including the ON ERROR GOTO statement at the beginning of the program slice to be monitored. Once error trapping has been enabled, if an error occurs, an automatic branch is taken to the statement number specified when the trap was set.

The EXTENDED BASIC error-processing routine uses system variables ERR and ERL to report the error and aid in proper recovery. ERR returns the error code of the most recent error. ERL contains the line number of the statement that was executing when the error occurred. Both variables can be used by the programmer in building error handling routines.

## Invoking a Subroutine

Unlike other high-level languages, BASIC subroutines do not begin with a special statement; any executable statement can be used as the first statement of a subroutine. They are called by line number and must terminate in a RETURN statement. A useful practice is to assign distinctive line numbers to subroutines. For example, the main program might use line numbers up to 300, and intended subroutine statements might start at line numbers 1000 and 2000, respectively. Below is the source for a program that invokes a subroutine, consisting of line 1000. The program prints "Execution continuing" several different times.

```
10     PRINT "Hi there."
20     GOSUB 1000
30     PRINT "At line 30."
40     GOSUB 1000
50     GOSUB 1000
60     GOTO 32767
1000   PRINT "Execution continuing."          &
             \RETURN
32767  END
```

When executed, this program produces the following output:

```
Hi there.
Execution continuing.
At line 30.
Execution continuing.
Execution continuing.
```

## Chaining Programs

The CHAIN statement should be used only when a program is too large to load into memory and run in one operation. The oversized program can be broken into two or more separate programs, with a CHAIN statement to call other programs into memory after the first has run. The syntax for the CHAIN statement is:

CHAIN ‹newprogfile› [LINE ‹lineno›]

‹Newprogfile› is a string expression (enclosed in double quotation marks or given in a string variable) that contains the Tandem filename in which the second program can be found. ‹Lineno›, when given, specifies the line number of the second program at which execution is to begin. The example below shows how to chain the current program with a second program in "$basic.subvol1.file2" and begin execution at line 30 of file2:

```
20  CHAIN "$basic.subvol1.file2" LINE 30
```

## Functions

EXTENDED BASIC has many built-in functions which are used to perform numeric and string operations on user-supplied arguments. Standard functions have a type (integer, string, or default numeric), which is indicated by the last character of the function name. Some of the more commonly used built-in functions are summarized in Table 4.

EXTENDED BASIC allows users to determine their own functions, and reference them in the same way they do standard functions. A user-defined function name consists of the letters FN, followed by up to 27 characters, followed by a single character determining the type of the function. A function is defined by a DEF statement, and if the function contains more than one line, it is terminated with an FNEND statement. Some example functions are shown in Figures 12 and 13.

**Figure 12**

```
 5   DEF FNCAPITALIZE$ (a$) = EDIT$(a$, 32)    !capitalize letters
10   DEF FNINCREMENT% (i%) = i% + 1   ! increment an integer by 1
```

**Figure 13**

```
1       !  ** Determine the number of roots to a quadratic equation
5       ON ERROR GOTO 19000
8       !  ** Here's the function definition...
9       !
10      DEF FNROOT$ (a,b,c)
20      discr = SQR((b*b) - (4*a*c))
30      IF discr > 0 THEN FNROOT$ = "2 roots."                            &
              ELSE IF discr = 0                                          &
                     THEN FNROOT$ = "1 double root."                     &
                     ELSE FNROOT$ = "imaginary roots."
40      FNEND
41      !
49      !  ** Here's the regular program.
50      INPUT "What are the coefficients for a, b, c", A, B, C
60      answer$ = FNROOT$(a,b,c)
70      PRINT "Answer is", answer$
80      GOTO 32767
19000   REM Error routine begins here.
19001   REM Err = -28 means negative value supplied to sqr function
19010   IF err = -28 THEN answer$= "imaginary roots." \RESUME 70
19999   ON ERROR GOTO 0   ! Unexpected Error.
32767   END
```

**Figure 12.**
*Single-line functions. These functions can be referenced in exactly the same way as built-in functions.*

**Figure 13.**
*Using a multi-line function (FNROOT$) in a program.*

## Examples

Figure 14 contains an EXTENDED BASIC program that sorts 10 user-input numbers according to a bubble-sort algorithm. A bubble sort consists of consecutive comparisons of two values at a time. Each time the

**Figure 14**

```
10      !    ** Bubble-sort Program **
11      ON ERROR GOTO 19000
20      DIM x(10)
30      FOR i% = 1 TO 10                                        &
            \INPUT "Next number", x(i%)
40      NEXT i%
50      FOR j% = 1 TO 10
60          FOR k% = j% +1 TO 10
70              IF x(k%) < x(j%) THEN                           &
                  temp = x(j%) \x(j%) = x(k%) \x(k%) = temp
80          NEXT k%
90      NEXT j%
100     FOR i% = 1 TO 10                                        &
            \PRINT, x(i%)                                       &
        \NEXT i%
19000   ON ERROR GOTO 0      ! Unexpected error.
32767   END
```

**Figure 14.**
*Bubble-sort program. A bubble sort consists of consecutive comparisons of two values at a time. Each time the second value is smaller than the first, the values are swapped. Thus, the smaller values are said to "bubble up" to their correct position in the sequence. After 10 times through the major loop (lines 50 through 90) to sort 10 values, the numbers are in correct ascending order.*

second value is smaller than the first, the values are swapped. Thus, the smaller values are said to "bubble up" to their correct position in the sequence.

The program in Figure 15 is a simple phonebook application using EXTENDED BASIC. The record stored in the file *nfile* consists of a 10-character field for a last name, a 10-character field for a first name, and a 20-character field for a phone number. The program provides the user with four functions:

- Deletion of a record.
- Insertion of a new record.
- A full list of the phonebook contents.
- A HELP facility.

Each is specified by the user as a single-letter option.

**Table 4.**
Commonly used built-in functions.

| Function | Action |
|---|---|
| ABS | Returns the absolute value of the specified expression. |
| CLK$ | Returns the current system time. |
| DAT$ | Returns the current system date. |
| DEVICEINFO | Returns the device type and subtype for a specified filename. |
| EDIT$ | Used to trim the parity bit, discard imbedded spaces, capitalize all alphabetic characters, change TABs to spaces, or convert other character strings. |
| ERR$ | Returns the error text corresponding to a specified error number. |
| LEFT$ | Extracts a substring from the beginning of a string. |
| LEN | Returns the number of characters in a string. |
| MID$ | Extracts a substring from the middle of a string. |
| POS | Returns the starting position number of a specified substring within a string. |
| RIGHT$ | Extracts a substring from the end of a string. |
| RND | Returns a pseudo-random number. |
| SQR | Returns the square root of a specified expression. |

**Figure 15**

```
10      ON ERROR GO TO 19000
100     MAP(n.phone)   n.phone$ = 40
101     MAP(n.phone)                                                      &
                n.phone.lastname$  = 10,                                  &
                n.phone.firstname$ = 10,                                  &
                n.phone.phone$ = 20
120     OPEN "nfile" AS #1, INDEXED, ACCESS MODIFY, ALLOW NONE,           &
                 MAP n.phone, PRIMKEY n.phone.lastname$                   &

190     : Name                      Phone Number
191     : 'LLLLLLLL   'LLLLLLLLL   'LLLLLLLLLLLLLLLLL                     &

500     DEF fnparse% (allowed$, input.char$)                              &
              \val$ = EDIT$ (input.char$, 32)    !capitalize              &
              \fnparse% = POS (allowed$, val$, 1)                         &
         \FNEND                                                           &

1000    REM   **  Determine operation user desires to perform   **
1010    INPUT "Option", op$                                               &

1020    IF LEN(op$) <> 1 THEN                                             &
              PRINT "Enter a single character only—CTRL/Y to exit."       &
              \GOTO 1010
1030    x% = fnparse% ("HDIL", op$)
1035    !                 H     D     I     L
1040    ON x% GOSUB  2000, 2100, 2200, 2300                               &
              \PRINT     \GOTO 1010                                       &

2000    REM                ** Help Routine **                             &

2010    PRINT "Valid options are"                                         &
              \PRINT "      D—Delete a record"                            &
              \PRINT "      H—HELP display"                               &
              \PRINT "      I —Insert new record"                         &
              \PRINT "      L —List phonebook"
2090    GOTO 1010                                                         &

2100    REM             ** Delete Record Routine **                       &

2110    INPUT "Lastname of record to delete", a$                          &
              \IF LEN(a$) <= 0 THEN GOTO 32767                            &

2120    a$ = EDIT$ (a$, 32)
2130    GET #1, KEY EQ a$              ! Find record
2140    PRINT "Record is"                                                 &
              \PRINT USING 190                                            &
              \PRINT USING 191, n.phone.lastname$, n.phone.firstname$,    &
                     n.phone.phone$                                       &
              \INPUT "Should this be deleted (Y/N)", yn$
2150    IF EDIT$(yn$, 32%) <> "Y" THEN GOTO 1010
2160          DELETE #1
2170          PRINT "Deletion complete."
2190    GOTO 1010                                                         &

2200    REM             ** Insert Record Routine **                       &

2210    INPUT "Last Name", n.phone.lastname$                              &
              \INPUT "First Name", n.phone.firstname$                     &
              \INPUT "Phone Number", n.phone.phone$                       &

2220    n.phone$ = EDIT$(n.phone$, 32%)       ! capitalize entire record
2230    PUT #1
2290    GOTO 1010                                                         &

2300    REM             ** List File Routine **                           &

2310    FIND #1, KEY GE " "       ! Find first record                     &

2320    GET #1                                                            &
              \PRINT USING 191,                                           &
                     n.phone.lastname$, n.phone.firstname$, n.phone.phone$ &
              \GOTO 2320                                                  &

19000   REM             ** Error Routines **                              &

19010   IF ERR=1  AND ERL=1010 THEN RESUME 32767       ! CTRL/Y           &

19020   IF ERR=1  AND ERL=2130 THEN                    ! EOF on DELETE    &
              PRINT "Record not found."     \RESUME 32767                 &

19030   IF ERR=10  AND  ERL=2230 THEN                 ! Record already exists&
              PRINT ERR$(ERR)  \RESUME 2190                               &

19040   IF ERR=1  AND ERL=2320 THEN RESUME 1010   !eof on LIST            &

19998   ON ERROR GOTO 0        ! If error detected while in trap routine,
19999                          !  then return to system error handler.
32767   END
```

**Figure 15.**
*Simple phonebook application. The program allows for the deletion or insertion of records, as well as the ability to list all records in the file "nfile." Each record corresponds to one individual, containing the person's first name, last name, and phone number.*

## Conclusion

Tandem EXTENDED BASIC has a variety of features. The interpreter offers a rich repertoire of commands, allowing the user to create source files; list and update files; interpret and run files; and store files in source, pseudo-object, or object format. While EXTENDED BASIC statements can be used to construct a simple program to implement a particular algorithm, they can also be used to construct large programs consisting of many subroutines, user-defined functions, and chained programs. Error handling can also be done within the language. The EXTENDED BASIC I/O facilities offer simple ways to communicate between a program written in EXTENDED BASIC and terminals, processes, predefined data, or permanent files. In spite of all these capabilities, Tandem EXTENDED BASIC remains a relatively simple programming language to learn.

**Jim Meyerson** has been a systems analyst in Tandem's Customer Application Support Group since joining Tandem in June 1982. He has been the EXTENDED BASIC Support Team Coordinator since September 1983, having previously served as a consultant for another language to a Tandem software development group. Jim taught languages and operating systems courses for another computer vendor before joining Tandem.

**References**

*ANSI Standard for Minimal BASIC.* X3.60–1978. ANSI, Inc.

*ENSCRIBE Programming Manual.* April 1983. Tandem Computers. version B00.

*Tandem EXTENDED BASIC Reference Manual.* December, 1983. Tandem Computers. version A00 (part #82380).

**Acknowledgments**

Much of the information in the tables and syntax descriptions in this article was taken from the *Tandem EXTENDED BASIC Reference Manual,* written by Angel Grindon, Software Development. Angel also offered many helpful suggestions for the article, as did Chris Sheedy of Tandem Software Development, and Geoff McDonald of Management Information Systems Pty. Ltd. Geoff wrote the phonebook application shown in Figure 15.

TANDEM
NonStop™ Computer Systems