Copyright © 1983 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 62, No. 3, March 1983
Printed in U.S.A.

Traffic Service Position System No. 1B:

Software Development System

By T. G. HACK, T. HUANG, and L. C. STECHER

(Manuscript received June 30, 1982)

This article describes the Software Development System for the Traffic Service Position System No. 1B (TSPS No. 1B). It discusses the modern, multicomputer software generation and test facilities that were provided to concurrently support both C-language and emulated, assembly-level software development. The computing environment, software generation and test tools, and standard development process that were developed for the TSPS No. 1B provide a rich, robust programming environment for future network operator services.

I. INTRODUCTION

The development and testing of the software for the Traffic Service Position System No. 1B (TSPS No. 1B) was a complex undertaking. In addition to emulating the existing TSPS No. 1 assembly-level program, 3B20 Duplex Processor (3B20D) native-mode (C-language) software was developed to interface to the Duplex Multi-Environment Real Time (DMERT) operating system and to provide the necessary system integrity functions for the TSPS No. 1B.[1,2]

To support software development and testing in this mixed emulation/native mode, it was recognized early in the project that a robust Software Development System (SDS) and a rigorous set of standard software development procedures (or methodology) were necessary. Thus, basic requirements for tools, documentation standards, and control procedures were established for the software development environment for the TSPS No. 1B. These requirements specified that:

(*i*) A simple, interactive user interface to the SDS must be developed. Where possible, commands and procedures for emulation or

native-mode software development should be the same to minimize the complexity of the programming task.

(*ii*) The SDS for emulation-mode software development must be upwardly compatible from the existing TSPS No. 1 SDS. In parallel with the development of TSPS No. 1B, new generic features were being developed and deployed for TSPS No. 1 (such as Automated Calling Card Service).[3] To offer universal service, these features would also have to be concurrently emulated on the TSPS No. 1B.

(*iii*) Complete software change procedures and tools must be established to support the parallel development of TSPS No. 1 and TSPS No. 1B. These procedures and tools would have to support the early development phase of the project when programmers were initially developing code, as well as later phases of the project when system testing is converging to a certified, production software release.

(*iv*) Finally, a set of test facilities was needed to support the integration and system testing of the software. Special tools would be required to test the emulated, assembly programs, as well as the high-level, native-mode software executing under the DMERT operating system.

The following sections of this article describe how the TSPS No. 1B Software Development System was implemented to meet these requirements.

II. COMPUTING ENVIRONMENT FOR TSPS SOFTWARE DEVELOPMENT

Before discussing the specific software generation tools and test facilities of the TSPS No. 1B SDS, this section describes the computing environment for TSPS development and gives an overview of the development system.

2.1 Overview

TSPS No. 1B software development is supported by a multiprocessor computing system. The system consists of five computers: one remote IBM 3033 processor located at Bell Laboratories in Columbus, Ohio; and four Digital Equipment Corporation (DEC) PDP-11/70 minicomputers co-located with the TSPS development organization at Bell Laboratories in Naperville, Illinois. A variety of data links support interprocessor communications. Figure 1* illustrates the configuration.

The IBM processor and the two PDP-11/70 computers (labeled PSS1 and PSS2) constitute the Programmer Support System (PSS). It is on these systems that source modification, load building, and

* Acronyms and abbreviations used in the figures and text are defined at the back of this Journal.

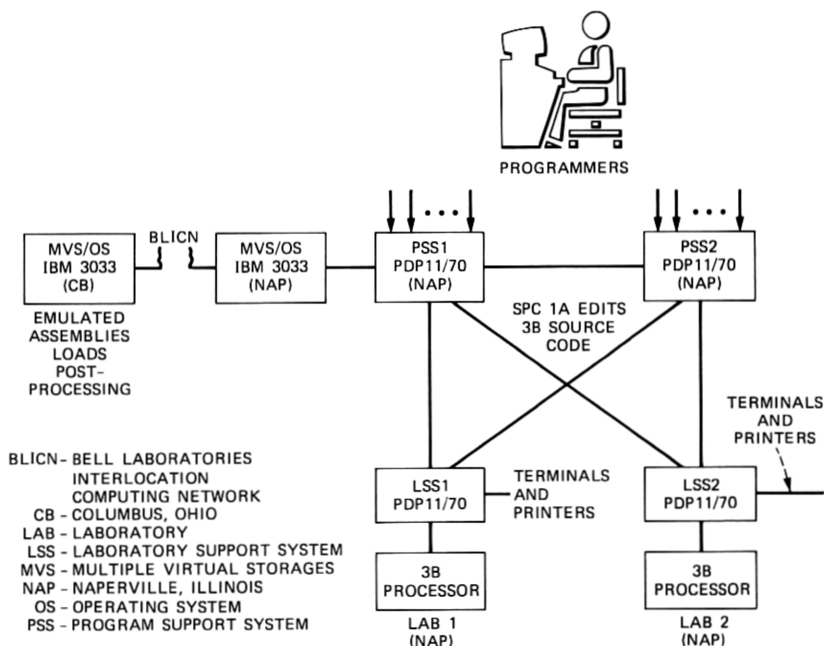

Fig. 1—Software Development System for TSPS No. 1B.

software administration are done. The remaining two PDP-11/70 computers reside in the TSPS system laboratories and are part of the Laboratory Support System (LSS). Each is paired with a 3B20D Processor and is used to support developer, integration, and system testing for 3B20D-based generic programs.

2.2 Local interactive development environment

TSPS programmers use the PSS1 and PSS2 computers to modify source files and create executable versions of software. These computers run the *UNIX** time-sharing operating system, which provides a general-purpose, multi-user, interactive development environment.[4] The *UNIX* operating system has a number of characteristics that make it attractive for software development. First, it is extremely easy to learn and use. The command-line interpreter (the "shell"), the interactive editor, and the hierarchical file system are particularly simple in nature. Second, the shell and C-language programming environment provide powerful and flexible facilities to create and combine software tools to support the generation and administration of software. Many of the tools used in TSPS have been created in this

* Trademark of Bell Laboratories.

environment. Finally, the Programmer's Workbench facility,[5] available with the *UNIX* operating system, offers:

(*i*) A sophisticated document preparation system to prepare documentation supporting the development process

(*ii*) A Remote Job Entry (RJE) subsystem that transmits jobs to other computing systems and returns output to appropriate users

(*iii*) A complete Source Code Control System (SCCS) for controlling and maintaining multiple versions of source code.

The software development is partitioned so that C-language developers use the PSS1 system and emulated, assembly-language developers use the PSS2 system.* System load building makes use of both PSS1 and PSS2. C-language development is solely supported on the PSS1 system. Emulated development, on the other hand, is accomplished through the interaction of the PSS2 system and the remote IBM 3033 processor.

2.3 Remote batch environment for emulated development

TSPS emulated software is developed in a part interactive, part batch-oriented computing environment. Much of the SDS for emulated code has been upgraded and carried over from the previously existing batch-oriented SDS for the TSPS No. 1.[6] The PSS2 system was added to the previously existing SDS for the TSPS No. 1 and provides the front-end user interface to the SDS for emulated programs. When the PSS2 system was added, the programmers gained a modern set of interactive commands that allow them to initiate the various software generation jobs for emulated code. Developers log into PSS2 and modify "edit files" that contain edit statements to be applied against the emulated source files, which are stored on the remote IBM system. The application of edits and all software-generation steps run under the Multiple Virtual Storages (MVS) operating system in a batch mode on the IBM processor.

For example, once a programmer has finished modifying the edit file on the PSS2 system, a single swap command can be issued to construct a Job Control Language (JCL) script designed to invoke, through the *UNIX* RJE facility, a remote emulated assembly on the IBM processor. The job file is sent via the Bell Laboratories Interlocation Computing Network and queued to be executed on the IBM/MVS system at Columbus. When the assembly is complete, the listing file is printed locally. Commands similar to swap are provided for link editing and other functions. These commands are described in more detail in Section III.

---

* Subsequent references to "emulated, assembly" language will be shortened to "emulated" language.

From the programmer's point of view, all activity takes place on the PSS2 system. The commands that create the remote batch jobs look very much like local commands. Since required output data (tapes, files, or listings) are returned to the local users, the use of the IBM/MVS system is transparent to the user.

2.4 Local interactive environment for C-Language development

The PSS1 system provides the computing environment for C-language program development. It can support as many as 36 simultaneous interactive users, although prime-time usage is typically about 25 users. For the most part, terminals are connected through a dial-up arrangement. There are, however, a few "hard-wired" terminal connections into the TSPS System Laboratories to guarantee access to programmers using laboratory test facilities.

Software tools that support C-language development on the PSS1 system include: the text editor, the 3B20D Software Generation System (3BSGS), the Change Management System (CMS), and the Source Code Control System. The text editor is used to modify C-language source files. The 3BSGS is a cross-compilation and link-editing system that generates executable files for the 3B20D Processor from source code. CMS and SCCS control and track the development process and provide the mechanisms to maintain multiple versions of source and object files.

From the programmer's perspective, the development scenario is quite simple. A source file is retrieved from SCCS, modified with the editor, compiled to produce an executable file using the 3BSGS, and data-linked to the Laboratory Support System computers for testing in the TSPS System Laboratories. With the exception of testing, all work takes place interactively on the PSS1 system.

2.5 Laboratory support processor

The support processor used in the TSPS Laboratory Support System (LSS) is a PDP-11/70 computer running under the *UNIX* operating system. The flexible environment of the *UNIX* operating system can support multiple testers simultaneously. For TSPS development, there are two system laboratories that are used to create the laboratory environment for field support and to enable realistic system testing of TSPS software. Section V describes the LSS in more detail.

2.6 Networking facilities

As we can see in Fig. 1, the processors within the TSPS computing environment are interconnected by a variety of data links that create a reliable, secure computer network. Each PSS computer is connected by two separate links to the other PSS computer and the two LSS

computers. One of the connections is a 9.6-kb link designed to be used with the cu command. (The cu command gives a user on one system the appearance of being logged into another.) Commands can be executed on the remote system and files can be transferred in both directions. The second type of connection is a 56-kb link that uses DEC DMC-11 hardware as the primary intercomputer file transfer mechanism. It can support effective file transfer rates of up to 2000 bytes/second. The remaining data link is a component of the Bell Laboratories Interlocation Computing Network connecting the Naperville and Columbus locations. The primary connection is established through 56-kb private lines between two IBM 3033 processors running the MVS operating system, the remote IBM system at Columbus, and a local interface system. The two TSPS PSS computers are connected to the local IBM/MVS system with special-purpose 9.6-kb hardware. Using the RJE subsystem, batch jobs can be sent through the local IBM/MVS system to the IBM/MVS system at Columbus. The capability also exists to return files to the PSS system or generate local output tapes or listings.

III. SOFTWARE GENERATION TOOLS

3.1 Overview

The primary function of a software generation system is to transform symbolic source language statements into a format executable by the target processor. For TSPS No. 1B, the target processor is the 3B20D Processor. A software generation system is usually a collection of tools that perform this transformation in phases. The compilation (or assembly) phase translates the source module containing symbolic instruction and data statements into machine-readable form, called a "relocatable object file." This file also contains symbol definitions and relocation information to be used in the linking-loading phase. The linking-loading phase enables separately compiled modules (relocatable object files) to be combined into a single executable unit, called a "load file." A final process-loading phase is used to transform the load file to a process-file format.

3.2 Emulated software generation

During the development of the TSPS No. 1B, special-purpose microcode was written for the 3B20D processor to emulate the Stored Program Control 1A (SPC 1A) assembly language used in TSPS No. 1. This allowed a large portion of the existing TSPS software to be transported to the 3B20D Processor without modification. In this approach, the emulated source statements were kept identical to the SPC 1A assembly-language statements. However, the object form (the machine-equivalent form) was modified to take advantage of the

powerful micro-instruction set of the 3B20D. This was done to maximize the real-time processing capability of the TSPS No. 1B system.

To ensure commonality between TSPS No. 1 and TSPS No. 1B assembly-level programs, the existing SPC 1A assembler and loader were retained. The new 3B20D object format for the TSPS No. 1B was produced by two new software generation tools—a "load post-processor" and a "listing post-processor." The load post-processor transforms an SPC 1A load file into a 3B20D load file, and the listing post-processor reformats the assembly listing to reflect the new, emulated instruction formats. In addition, many of the existing SPC 1A SDS tools were retained and significantly upgraded by the addition of the PSS2 system. Figure 2 shows the steps involved in emulated software generation.

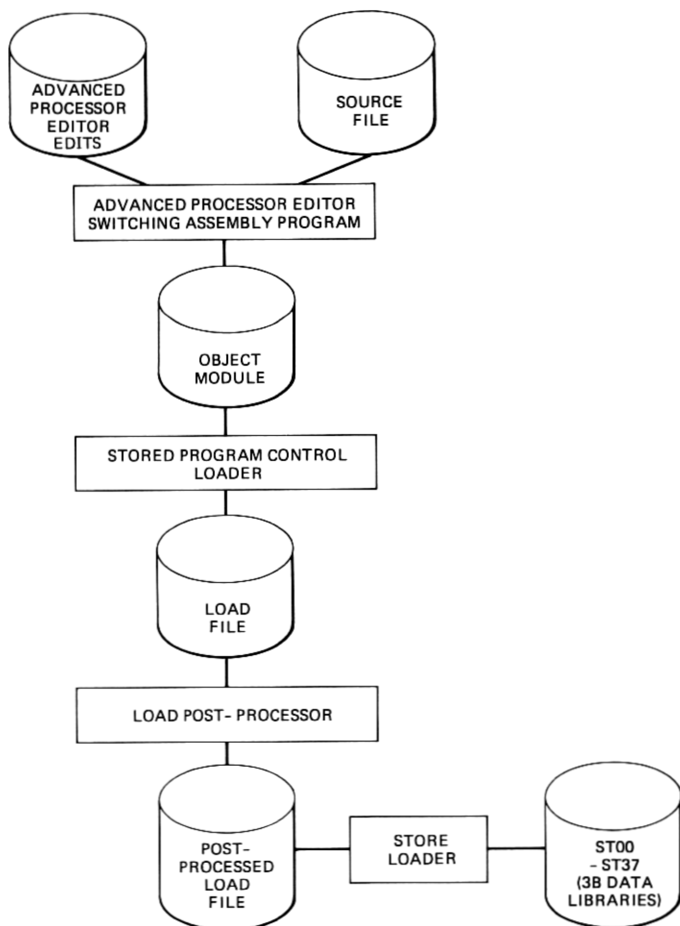

Fig. 2—Basic steps in emulated, assembly-language software generation.

3.2.1 The TSPS assembler and loader

The assembly process for TSPS emulated code combines both an editing and an assembly operation. Editor statements are prepared on the PSS2 system and transmitted to the remote IBM/MVS system. These editor statements are then processed and applied to the emulated source file by the Advanced Processor Editor (APE) running under MVS. APE is a simple, line-oriented editor with the basic "insert," "replace," and "delete" functions. It is specifically designed to work with the assembler, passing the edited source as input to the assembly process.

The assembly operation is performed by the powerful SPC-SWAP (Switching Assembly Program) assembler,[7,8] which also runs under MVS. Besides performing the standard source-to-object conversion, it has a sophisticated macro capability and a variety of useful pseudo-operations for controlling listing format and establishing symbol definitions. In addition, there is a mechanism for creating and maintaining special-purpose "library" files. Library files of symbol or macro definitions can be created in one SPC-SWAP run, then later accessed by subsequent source module assemblies for the purpose of symbol or macro resolution. This is not only a convenient mechanism for sharing global symbol definitions between source modules, but also allows a single source file to be assembled in different environments, resulting in different object modules. The latter technique is used in TSPS to develop code for multiple generics from a single source file and is discussed further in Section 4.1.

To perform an assembly, TSPS developers invoke the swap command on the PSS2 system. This command creates a Job Control Language script, which is executed on the remote IBM/MVS system. The primary outputs of SPC-SWAP are an object module, which is retained on the IBM file system, and an assembly listing, which is printed locally.

After the assembly process, the next step in producing an executable file for the 3B20D Processor is to combine the SPC-SWAP object modules for a given generic using the SPC loader. This loader uses a special set of control statements that specify what areas of emulated address space are available for loading object modules, which modules are to be loaded, and, optionally, what their load addresses are. The output of the SPC loader is a file or magnetic tape containing the relocated, fully bound, generic load file. At this point, however, the load file is suitable only for loading into an SPC 1A used in TSPS No. 1. Further post-processing (described in Section 3.2.2) is then done to produce a 3B20D-compatible format.

In addition to a full generic load, the SPC loader also has the capability to produce what is called a "partial-load" file. During the

generation of a full generic load, the SPC loader is directed to create a special file on the IBM/MVS system called a HISTORY. This HISTORY file contains the information necessary to completely characterize the generated load file. This includes the load image itself; the names, starting addresses, and sizes of the object modules; free space; and all external symbol definitions and references. Once a HISTORY file has been created, any TSPS developer on the PSS2 system can reassemble a selected subset of generic source modules and issue the ppload command to build a partial load on the IBM/MVS system. The items input to the ppload command identify the HISTORY file and the reassembled object modules. The partial-load process constructs a new load image from these input files, then compares it to the image contained in the HISTORY file. The final output is a magnetic tape containing the changed instruction or data words. This tape, after post-processing, can be taken into the TSPS system laboratory and overlaid on the full generic load file. The partial-load capability is used extensively in the early stages of TSPS generic development. It is a very flexible and convenient means of creating a developer's private software version for testing.

3.2.2 Post-processing— format conversion from SPC 1A to 3B20D format

To convert from SPC 1A format to 3B20D format, a load post-processor was developed to transform the output of the existing SPC loader into the new 3B20D encoding. In addition, a listing post-processor was developed to convert program listings to account for the changes in the order encoding and also to provide address and data fields in hexadecimal, which is the native number base for the 3B20D.

The load post-processor accepts as input a HISTORY file produced by the SPC loader. The HISTORY file contains not only the actual load image but also specifies whether an SPC word corresponds to a program or data instruction. Each 40-bit (double-word) SPC program instruction is translated into a 64-bit, 3B20D encoding (two 32-bits words) as required for the emulation.[2] The reformatting that is done depends on the SPC operation code and can completely rearrange the fields in an instruction. The translation algorithm is encoded into a set of common routines that are shared by the load and listing post-processor. This ensures that listing and load translations remain synchronized. Both the load and listing post-processors are run as batch jobs on the IBM/MVS system and are initiated remotely from the PSS2 system.

3.2.3 Creating a DMERT process from an emulated load file

The TSPS process is a key component of the TSPS No. 1B system since it has primary responsibility for call-processing functions.[2] It is

a DMERT kernel process made up of both C-language and emulated software. The C-language portion is built in the standard fashion using the 3B Software Generation System (see Section 3.3.1). The emulated portion is stored in DMERT data libraries and appears as pure data to the C-language software generation process. The data libraries are stored as separate files on the 3B20D disk and are linked into the TSPS process address space at process creation time. There are 32 data libraries for emulated software, each being one segment (128K bytes) in size.

3.2.4 Overwrite generation

In the later stages of the development of a generic, rigorous change procedures are introduced to tighten control over changes to the software. This is known as the "frozen" mode of development and begins normally with the start of system testing.[9] In frozen mode, changes in emulated code are applied in overwrite form rather than by full reassembly and linking of source modules. An overwrite consists of just those program and data instructions that are being modified in a program change. Thus, an overwrite tends to be small in size and the impact of the change is local rather than global. Overwrites are usually inserted into a stable program version and tested one at a time in a cumulative fashion. In this way, the generic program evolves in a controlled way with each change being tested before the next is applied.

3.3 C-Language software generation

During the lifetime of the TSPS No. 1 system, all software development was done in assembly language. However, for the introduction of the TSPS No. 1B, as well as for the development of future operator services, significant portions of the new software will be developed in the C programming language. Thus, a new set of software tools was implemented to support the native-mode development process for the TSPS No. 1B.

3.3.1 The 3B Software Generation System

The 3B Software Generation System (3BSGS) is a collection of software tools available with the 3B20D Processor.[10] These tools transform C-language source code into object files that can be loaded into the 3B20D disk and executed under the DMERT operating system. The main constituents of the 3BSGS are the C compiler (3bcc), the 3B20D assembler (3bas), the 3B20D link editor (3bld), and the 3B20D process loader (3bldp). These programs are designed to execute in the environment of the *UNIX* operating system.

The C compiler accepts C-language source files as input and trans-

forms them into relocatable object files. The 3B link editor, 3bld, can combine several object files into one by relocating program and data instructions and resolving externally defined symbols. It also provides the mechanism to resolve references to library routines. (Libraries are collections of precompiled modules that typically contain commonly used routines that are shared among many modules.) For TSPS No. 1B, the output of 3bld is passed to the 3B20D process loader for final processing.

The 3B20D process loader, 3bldp, takes as input a collection of relocatable object files produced by 3bcc or 3bld. By using a special-purpose specification file, it constructs an output, called a process file, which has a format appropriate for execution under the DMERT operating system.

3.3.2 The Make Program

For TSPS No. 1B, a utility program, Make, is used extensively in conjunction with the 3BSGS during the software generation process. The purpose of Make is to automate the construction of a process file when one or more of the source files on which it depends have been modified. The generation process is controlled by a special-purpose description file, frequently referred to as a "makefile."

The advantages of the Make program are that it:

(*i*) Ensures that a process file is constructed correctly in that no commands are accidentally forgotten or specified incorrectly

(*ii*) Regenerates only those files that have been affected by a change

(*iii*) Minimizes the programmer effort in building executable processes.

For these reasons, Make is used extensively in TSPS No. 1B development. For the initial TSPS No. 1B generic, there are approximately 50 makefiles used in building over 300 3B20D processes.

3.3.3 Dependence on DMERT header files and libraries

The TSPS C-language software runs under the DMERT operating system. The two chief software mechanisms that support communication between the TSPS application and the operating system are header files and libraries. Header files are typically used to define data structures that are shared between processes or shared between source files within a single process. For example, there is a header file that contains data declarations specifying the layout of a DMERT Process Control Block (PCB) for a supervisor or user process. Any source file that references the PCB will include this header file through the "#include" mechanism provided by 3bcc, by which the PCB data declarations are made available to the compilation process. Libraries,

on the other hand, typically contain commonly used functions or routines. References to libraries are resolved by 3bld during the software generation process. Libraries are usually organized on a functional basis. For example, the craft interface library contains functions and routines oriented towards applications involved with the handling of craft input messages or the generation of output messages.

The DMERT operating system contains a large number of header files and libraries to support TSPS development. With each DMERT release, applications receive not only the latest version of DMERT software, but also updated versions of the header files, library files, and the 3BSGS. These are installed on the PSS1 system to support TSPS development.

3.3.4 Linking C and emulated software

As previously mentioned, the TSPS process contains both C-language and emulated programs. The emulated code is allocated one megaword of the available two-megaword-process address space and contains the bulk of the TSPS call-processing software. Naturally, there is a need to transfer control between emulated and C-language programs. Since these programs require different versions of microcode, a processor "mode switch" is required. Two special assembly-language instructions, cale (call emulated) and smt (switch mode and transfer), were developed to perform this function. Cale is a native-mode instruction that effects a mode switch and a transfer to emulated address space. Smt is an emulated instruction that performs a similar function but in the reverse sense.

From a software-generation perspective, the interface between emulated and native mode is quite simple. Routines called through cale or smt are accessed through transfer vectors. An emulated program, called MADEP, contains a table of transfer vectors, one for each emulated routine called from native mode. To link from a native to an emulated routine, the developer defines a symbol, say etv, equal to the address of the transfer vector in MADEP. (These definitions are kept in a header file dedicated for this purpose.) The emulation routine can then be accessed by an "spcxfr(etv)" function call in a C program. Spcxfr is an IS25 assembly-language routine that executes a cale to effect the transfer.

To transfer from emulated to native mode, the developer defines a global symbol, say ntv, equal to the address of the transfer vector for the desired function. These transfer vectors are forced to specific locations by special control statements in the loader specification file of the TSPS process. Thus, their location is known to the programmer. The transfer is accomplished through the use of the smt instruction.

Although this emulated-native transfer mechanism requires some

manual coordination between emulated and C-language files, this effort is only required when a new routine (to be accessed from the opposite mode) is added. These routines can be modified at will without requiring the referencing programs to be regenerated. Since these special-purpose routines are added infrequently, emulated and C-language development can proceed, for the most part, quite independently.

3.4 Overwrite considerations for the C-Language environment

For C-language software generation, there is no direct analogue to an overwrite for emulated software. C-language frozen development involves full recompilation and link editing of TSPS processes. The effect of this on TSPS development is discussed in Section 4.3.3.

3.5 Building a 3B20D disk image

Thus far, the mechanisms for constructing DMERT processes files have been described. To complete the software generation process, it is necessary to build a 3B20D disk image containing the total collection of process files, special-purpose boot files, and equipment configuration information. To accomplish this, the DMERT operating system utilizes two major data bases: the Equipment Configuration Database (ECD) and the System Generation (SG) database. The ECD contains records describing the characteristics and status of all processor and peripheral hardware. The contents of the ECD specify the current hardware configuration of the system. This hardware status information is placed in a central database, the ECD, to eliminate redundant device information and to provide a unified approach to handling and accessing hardware configuration data. The SG database contains information specifying the structure of the 3B20D disk image as designed by the application. This includes operating system parameters, disk partitioning information, file system names and sizes, names and types of individual files, and the make-up of the operating system boot image.

A DMERT tool, 3bmkdsk, provides the capability to construct a disk image from the ECD and SG databases, DMERT process files, and TSPS process files located on the PSS. This program generates three 1600-bit-per-inch (bpi) magnetic tapes containing 32M bytes of data. The image on tape is used to create a 3B20D disk.

IV. THE SOFTWARE DEVELOPMENT PROCESS

The development of the TSPS No. 1B with many software tools and literally thousands of files requires a rigorous set of procedures (or methodology) to allow the development of software to proceed in an

orderly and controlled fashion. This section describes the methodology employed for TSPS No. 1B development.

4.1 General support environment of TSPS

At the source-language level, the initial TSPS No. 1B generic program comprises approximately 350 emulated source files and 380 C-language source and header files. In executable form, the emulated software amounts to about 2.3M bytes of program. This excludes scratch data areas and office-dependent data (describing the particular line and trunk arrangements of an office), which also reside in the emulation address space. The C-language source is transformed through software generation tools into over 300 3B20D process and data files.

To control development on this large collection of software modules, official and test versions of both emulated edit files and C-language source files are maintained under the Source Code Control System. SCCS is a collection of programs that can store and retrieve multiple versions of a file in a space-efficient manner. This not only maintains a history of changes but allows the retrieval of various versions of source code that, for example, might represent a recent release, the current official software, software under system test, etc.

In addition to using SCCS for source control, TSPS also makes use of a "featuring" concept. Featuring allows developers to maintain a *single* source file regardless of the number of program generics under development. In featuring, lines added, deleted, or replaced in a source file are bracketed by feature-control directives which, during the assembly of the source file, direct the assembler either to assemble or ignore bracketed source code. The feature-control directives used in TSPS emulated development are:

(*i*) INFOR feature-expression
(*ii*) OUTFOR feature-expression
(*iii*) ENDFOR feature-expression.

Feature-expression is a Boolean combination of feature names. In an assembly, each feature name, and consequently each feature-expression, evaluates to a true or false value. If the feature-expression associated with an INFOR directive is true, the source lines between the INFOR and corresponding ENDFOR are assembled. For OUTFOR, if a feature expression is true, it causes the bracketed source lines not to be assembled. A set of feature expressions is associated with each generic. In this way, a single source file can generate multiple distinct object files, each associated with a particular generic program under development. The featuring syntax used for C software is different, but has the same basic capabilities.

There are two modes of development used in TSPS. The "non-

frozen" mode is the first stage in the development process when most of the new feature software for a generic is written. The intent is to give the programmers as much freedom as possible in writing and testing their code. When most of the new feature development is complete, the "frozen mode" is entered. In this mode, tight control is exercised over changes that are applied to the generic. Each change is documented and tested individually to assure a controlled evolution of the generic software. The sections that follow describe in some detail how development works in these two distinct environments.

4.2 Non-frozen-mode methodology

4.2.1 Emulated development

4.2.1.1 Creation and modification of source modules. In the non-frozen mode, programmers have a large degree of freedom in modifying existing source files or creating new ones. To create a new source file for emulated program development, the developer simply uses the interactive editor to input emulated statements. The new source file can be assembled, put into executable form by the partial load capability, and tested in the TSPS system laboratories. At the time of the next "base load" (see 4.2.1.3), the source file will be installed on the remote IBM/MVS system where all emulated source files are maintained.

Most non-frozen-mode activity involves modifying APE line edit files under the control and administration of the TSPS Subsystem for Non-Frozen Development (TSPNF).

TSPNF uses a strategy that allows multiple developer teams to work in parallel in a non-interfering way. The hierarchical nature of the file system is used to logically group edit files based on generic name, base-load name, and team name. Thus, several teams may have edit files affecting the same source file. Individual team members can create edit files within the team directories via the TSPNF commands, nfget and nfput. The nfget command is used to retrieve from the TSPNF directory structure, for the purpose of editing, the edit file for a source module corresponding to a particular generic, base load, and team combination. If the TSPNF structure has no edit file for the particular team, nfget retrieves the latest official version of the edit file. The edit file is placed in the user's current directory (work area). Once the file is retrieved, the programmer can modify it as described by interactively adding, deleting, or replacing source lines or feature control directives. Once the edit file has been changed to the programmer's satisfaction, it can be returned to TSPNF for safekeeping by the nfput command.

4.2.1.2 Generation of private versions for testing. The PSS2 interface for emulated program development contains a number of commands

to be used for the generation of IBM Job Control Language (JCL) scripts and for the submission of these scripts to the remote IBM/MVS system for execution. Two of these commands, swap and ppload, play a key role in generating software for testing in the non-frozen mode of development. The swap command is used to generate and submit the JCL for a SPC-SWAP assembly.

The swap command allows the programmer to assemble a new source file that has been created on PSS2 or, in the case of a program whose source is already resident on the IBM/MVS system, to specify the name of an APE edit file to be applied to the program source prior to assembly. Options are provided to retrieve the edit file from the user's current directory, the TSPNF team directory structure, or the official set of line edit files.

Regardless of the source of the input, a *generic-name* parameter is used to establish the proper environment for the assembly. This includes setting up references to the correct SWAP symbol and macro libraries, selecting appropriate feature names, and specifying generic-dependent listing-format options. Typical output files from the swap command are the assembly listing file and an object module. These both reside on the remote IBM file system with the object module placed in a special team partitioned data set (a single data set containing a collection of object modules for a specific development team). The listing file can be printed locally in original or post-processed format.

The ppload command provides the programmer interface to the partial load capability described in Section 3.2.1. This command will generate a partial load run using the HISTORY file from the latest base load (see Section 4.2.1.3) and all of the object modules in a specified set of team data sets. The result is a file on the IBM/MVS system in post-processed format that contains just the program differences introduced by the team object modules. Through the use of an optional parameter, this partial load file can be written to tape. This then provides a convenient mechanism to create a private, team version of software that can be taken into the system laboratory, overlayed on the current official version from the last base load, and tested.

The non-frozen development scenario from the programmer's point of view is quite simple. An edit file for the source file to be modified is retrieved with the nfget command, and changed with the interactive editor. Private developer or team test loads can be built by use of the swap and ppload commands. Once the edits have been tested, the edit file can be returned to the team TSPNF structure with the nfput command. All programmer activity with the exception of the testing itself occurs on the PSS2 system.

4.2.1.3 Submission and integration of software changes. Periodically, during the process of developing a new generic, newly developed software is integrated with the latest approved version of the generic program and tested. This process is called the "base load" process because it results in a new, independently certified program load that serves as the base for future development. For a base load, each team leader is responsible for submitting a complete description of the team input. This information includes the list of team edit files to be incorporated into the base load, the features that are completed with this submission, and other appropriate documentation as needed. Using this input, a new software version is constructed and turned over to the TSPS Integration Group for certification in the system laboratory. When certification is complete, the software is made available to the developer community.

4.2.2 C-Language development

Thus far we have described the non-frozen development environment for emulated programs. This section addresses the C-language development environment. The Change Management System (CMS) plays a key role in both frozen and non-frozen C-language development in TSPS. To understand how C-language development proceeds, it is necessary to grasp the basic concepts employed in CMS.

4.2.2.1 The Change Management System. CMS is a collection of *UNIX* programs aimed at controlling the activity of programmers, test-teams, and administrators engaged in C-language software development. In CMS, all development activity is tied to the notion of a modification request (MR).

An instance of CMS has three main components:

(*i*) A set of project source files maintained under the Source Code Control System (SCCS), described earlier. This allows multiple versions of a source file to be kept (for example, the current official version, a version undergoing system test, and a developer's private version).

(*ii*) A relational database that maintains the status of MRs and relates MRs to SCCS (version) identifiers. This not only provides the status of MRs but also permits the retrieval of source file versions that correspond to particular features or code changes.

(*iii*) A set of directory structures called *nodes*, each of which is identical in makeup (i.e., reflects the generic program's directory structure) and is capable of holding all source, object, process, and data files that constitute a generic program.

The use of CMS in TSPS C-language development will be the topic of the next several sections.

4.2.2.2 Creation and modification of source modules. CMS maintains

an official source repository that consists of one SCCS file for each source file used in a generic development. For TSPS, this includes C-language source files, header files, and makefiles. These SCCS files are stored in a project directory structure, a subtree of the hierarchical file system, organized so that files making up a single process typically are found in a single directory, while the files associated with related processes are grouped in directory structures that are subtrees of the overall project directory structure.

As previously mentioned, the SCCS files can contain multiple versions of the corresponding source file. To modify a source file, the programmer first extracts the latest version of the file to be modified from the SCCS file in the official repository. The programmer then edits the file with the text editor, builds an executable version of the affected process file, and tests the resulting product. When satisfied with the changes to the source file, the programmer returns the new version of the file to SCCS.

All source file change activity is associated with an MR number. Basically, the MR is a name under which a set of software changes are grouped. The programmer is quite free to use MRs as necessary during the non-frozen mode of development. Typically, the programmer will define an MR to represent a feature or subfeature being developed.

4.2.2.3 Generation of a private version of software for testing. The generation of executable process files from C-language source uses the 3BSGS and makefiles, as descibed in Sections 3.3.1 and 3.3.2. However, rather than using the make command, an enhancement of make, called build is employed by TSPS developers. Build is a software tool provided as part of CMS. It provides a very convenient mechanism to construct a private version of a process or set of processes. The programmer executes a single command and only has to deal with those files that have been modified. Other required files are automatically shared from the official directory structure. This sharing not only saves file space, but also eliminates the need for time-consuming, error-prone copying of files.

In TSPS, each programmer maintains a private node for development. All programmers share the official node. The developer scenario is as follows:

(*i*) A command is issued to retrieve the source files to be modified.

(*ii*) The retrieved source files are edited.

(*iii*) Executable versions of the software are constructed using build.

(*iv*) The executable files are transported to the system laboratory for testing.

(*v*) When changes are complete, the source files are returned to SCCS.

There is little chance for error during software construction since build

automatically generates the commands to reprocess any and all files that have been affected by the editing. This eliminates the manual effort to reprocess files, the potential errors caused by forgetting to reprocess files, and the overhead of unnecessarily reprocessing files.

4.2.2.4 Submission and integration of software changes. In the non-frozen mode, as developers complete testing of new software associated with an MR, the code is submitted by executing the CMS submit command. This changes the status of the MR in the CMS database from "active" to "submitted." In addition, documentation specifying the feature(s) covered by this submission and the source and process files affected is turned over to the TSPS Integration Group.

As with emulated software, (and usually at the same time), C-language software undergoes a "base load" process. After the load is tested and certified, the official node is updated with all files in the test node and the MRs are marked "approved" and "integrated" in the CMS database. Non-frozen development can then proceed with the newly tested and approved official node.

4.3 Frozen-mode methodology

4.3.1 Overview

In the non-frozen mode of development, the emphasis is on giving the programmer freedom to make large-scale changes to existing software to facilitate new feature development. As system testing begins, TSPS software is placed under rigorous change control procedures. Problems are documented in the form of trouble reports, which are then carefully monitored. The developers submit correction reports (CRs), which include a description of the fix, as well as the software change itself. Each CR is tested individually against the existing approved software version and must be approved by a Change Review Committee with representatives from all involved project teams before it is officially incorporated in the generic.

The intent of this method is to evolve the software in a rigorously controlled manner. Thus, when operational problems arise, they can be localized more easily to a particular area of software. With this approach, a high degree of confidence in the evolving software product is realized.

4.3.2 Emulated development

From a programmer's point of view, the scenario for creating and testing a correction in emulation code in frozen mode is similar to developing code in the non-frozen mode; however, the unit of change is substantially different. In non-frozen mode, the unit of change was the partial load. In frozen mode, it is the overwrite described in Section 3.2.4.

Special "patch" directives recognized by SPC-SWAP are placed by programmers in the overwrite source code to ensure that a change does not cause the entire object file to be relocated in memory. Replacement of program or data words is permitted, but only by instruction sequences of equal size. If a change is larger than the instruction sequence being replaced, a transfer to "patch" area is made. (Patch area is spare memory space reserved for this purpose.) In this way, object modules remain the same size and the number of memory locations whose contents change is minimized.

4.3.3 C-Language development

For C-language development, there is little difference between the frozen and non-frozen modes. The same tools are used and the same basic development scenario is followed. The main difference is that each MR represents an existing system problem or minor enhancement rather than a new feature. Therefore, the amount of software associated with an MR tends to be of a smaller quantity. In addition, as with emulated software, software change reports are generated and submitted to document the change and must be approved through the same process.

V. LABORATORY SUPPORT SYSTEM

The Laboratory Support System is a software testing system that enables users to test their programs in a TSPS system laboratory. It is generally concerned with the laboratory execution environment of TSPS programs.

5.1 System laboratory configuration

To provide sufficient test capabilities for TSPS development, there are two independent TSPS system laboratories. Generally, both system laboratories support parallel execution environments for TSPS programs. The configuration for a TSPS system laboratory is shown in Fig. 3.

5.1.1 System laboratory hardware configuration

The TSPS No. 1B is controlled by the SPC 1B, which consists of a 3B20D Processor and a Peripheral System Interface (PSI). The complete description of the SPC 1B is detailed in Ref. 11. In addition to the SPC 1B, a TSPS system laboratory also contains a set of TSPS peripheral units, operator consoles, and a Laboratory Support System.

5.1.2 Laboratory Support System

The Laboratory Support System is primarily used to support debugging on the SPC 1B, for controlling the operations of the SPC 1B

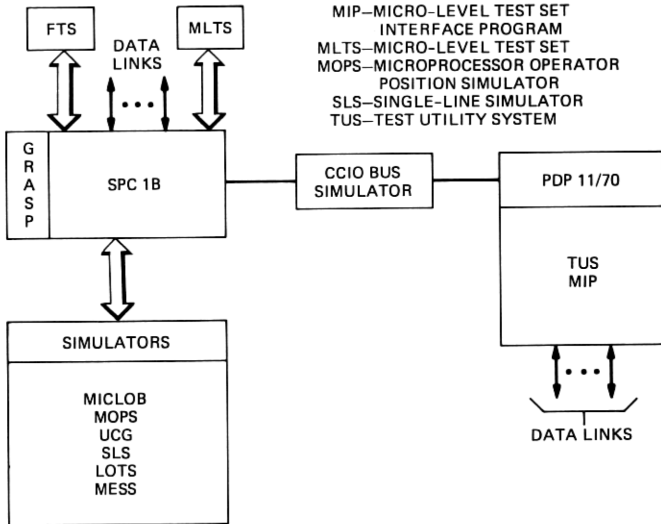

Fig. 3—Laboratory configuration for TSPS No. 1B.

in the test environment, and for loading the SPC 1B memory. The LSS consists of a PDP-11/70 support computer, special laboratory and generic utility systems, and TSPS call simulators, which support program testing. The PDP-11/70 is connected to the SPC 1B by a Central Control Input/Output (CCIO) bus simulator unit.[11]

There are four major debugging tools associated with a TSPS LSS. They are the Test Utility System, the Micro-Level Test Set (MLTS), the Field Test Set (FTS), and the DMERT Generic Access Program (GRASP). Each provides a distinct set of debugging capabilities.

5.1.2.1 Test Utility System. The Test Utility System (TUS) is the principal, high-level, debugging tool for the software developed for the SPC 1B. TUS provides symbolic access to data and a "C"-like utility language. It resides partially on the LSS and partially on the SPC 1B. The TUS support processor subsystem on the LSS performs TUS input and output processing and TUS system control. Users log into TUS on the support processor subsystem; this subsystem converts the user's utility commands into executable command groups that are then sent to the TUS test processor subsystem on the SPC 1B. The test processor subsystem executes under the DMERT operating system. Symbolic references are resolved on the LSS using symbol tables constructed on the PSS as part of the base load process.

As the TUS test processor subsystem on the SPC 1B receives command groups to be executed, requested break points or matchers are set up, and associated utility commands are processed. Any raw data that result from the utility commands are collected and sent back to the TUS support processor subsystem to be processed. Output to the user includes symbolic references and processed data. Commands are available to: read or write any memory location, display or send a message to a process or port, enable or disable a program trace, copy a file from the test processor to the LSS, start or stop a test process, and send an event or fault to a test process with associated data input.

Because of its symbolic access to data and its "C"-like utility language, TUS is a powerful debugging tool. In addition, TUS is an integral part of the system overwrite facility. TUS supports a data link that is capable of transferring files at a rate of 56K baud. Using TUS, overwrites are quickly transferred from the LSS to the SPC 1B when requested by the programmer or tester. More details about TUS can be found in Ref. 10.

5.1.2.2 Micro-level test set. The micro-level test set connects to the microbus of the SPC 1B and is used for direct access to 3B20D registers and memory. From a terminal connected to the LSS, simple commands can be sent to the MLTS to insert breakpoints. After a breakpoint fires, the SPC 1B is halted so that the tester can display or change the contents of registers and memory locations. Once this is done, the processor can be restarted. MLTS (unlike TUS) is a stand-alone tool that is completely independent of the operating system on the SPC 1B and can be used to debug at the kernel level. However, it does not provide symbolic testing capabilities and is used only for low-level program debugging.

5.1.2.3 Field test set. The field test set is a portable debugging tool capable of tracing the execution of processes without interfering with normal processor operations (Fig. 4). As such, it provides an effective debugging capability both in the system laboratory and operational field sites. It interfaces directly with the dual-access utility circuit in the 3B20D Processor to record information about the execution of processes in the SPC 1B. The tracing capabilities of the FTS are controlled by simple commands that set up matchers, trigger functions, and trace memory. The FTS can be used to perform a process trace, transfer trace, data history trace, simultaneous data history and transfer trace, function trace, and simultaneous data history and function trace. Like the MLTS, the FTS is completely independent of DMERT.

5.1.2.4 Generic Access Program. The final debugging tool used in the development of the TSPS No. 1B is the Generic Access Program (GRASP), which is a standard DMERT utility system resident on the 3B20D. It provides basic, non-symbolic capabilities to dump registers

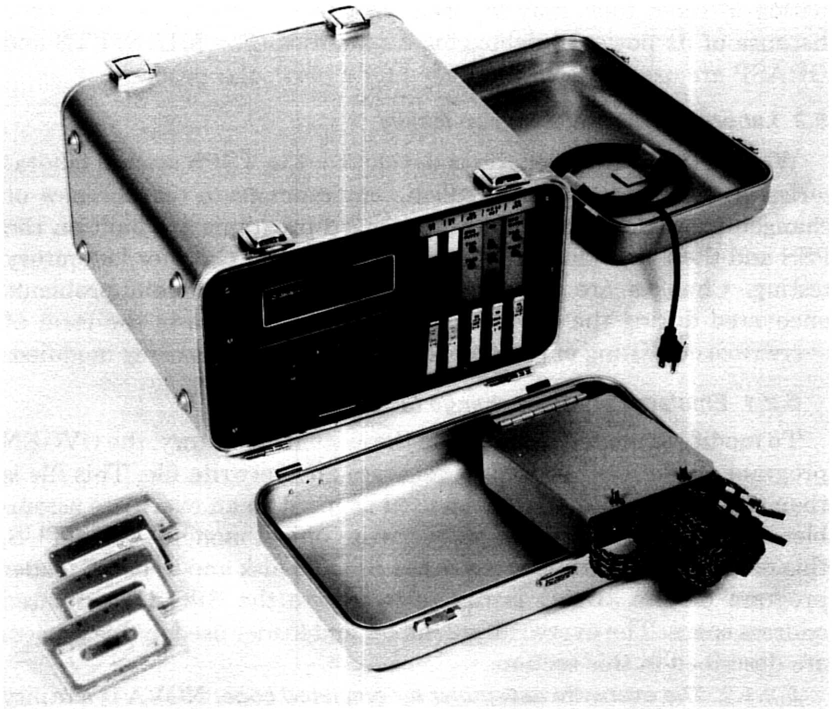

Fig. 4—The field test set.

and memory, and to set up breakpoints to stop program execution at desired points. Using the 3B20D dual-access utility circuit, GRASP also can trace the flow of execution of instructions of a process in a non-interfering fashion. A typical trace might record the "from address" and "to address" of every branch instruction executed. Additional information, such as the contents of the registers, may also be obtained. Because it is running on the same processor as the test processes and requires the support of the DMERT operating system, GRASP will generally interfere (like TUS) with the normal operations of a process and is not effective for kernel debugging.

5.1.2.5 Choosing the right debugging tool. As evident in the previous discussions, the four debugging tools used in the TSPS No. 1B development were designed for specific applications and have overlapping utility capabilities. TUS is a high-level, symbolic debugging tool. The MLTS is a low-level debugging tool. The FTS is a non-interfering debugging tool in the sense that normal instruction sequences and timing are not altered when using it. GRASP is a generic tool available on an operational 3B20D. Within a particular test session, a combi-

nation of these tools may be used. Whenever possible, TUS is used because of its powerful debugging capabilities. The MLTS, FTS and GRASP are used only when needed for a particular problem.

5.2 Laboratory software change facility

When a new TSPS generic is developed, the TSPS system laboratories are used to provide a realistic environment to test the new or changed programs. These new or changed programs are built on the PSS and then down loaded via the LSS to the SPC 1B for laboratory testing. Changes are often required to correct program problems uncovered during the lab session. These changes are in the form of overwrites consisting of program and data instructions being modified.

5.2.1 Emulated program change facility

To modify emulated programs in the system laboratory, the OVGEN program on the PSS is used to generate an overwrite file. This file is then transferred to the LSS and used as input to an overwrite assembler on the LSS to produce an overwrite object module. Using TUS, this module is then transferred to the SPC 1B disk and a special loader program on the 3B20D is run to overwrite the SPC 1B emulated address space. The overwrite assembler and loader used in this process are described in this section.

5.2.1.1 The overwrite assembler for emulated code. NOVA is a utility program on the LSS used to assemble overwrites for SPC 1B emulated code. The input to NOVA is the swap overwrite file produced by OVGEN, and the output is an absolute or relocatable object module and a relocation dictionary for the overwrite. The information contained in the relocation dictionary consists of pointers into the object file and addresses to be assigned by the loader. NOVA also produces overwrite and cross-reference listings to document the change in the laboratory.

5.2.1.2 The overwrite loader for emulated code. The loader program (NULOAD) is a relocatable loader for emulation code on the 3B20D. It provides a means by which NOVA-assembled overwrites are loaded into the TSPS process address space in memory. During each lab session, NULOAD automatically monitors the allocation of patch space and assigns patch space as needed by the overwrites. It allows the user to combine various overwrites assembled at different times and by different people without having one patch in one overwrite loaded on top of a patch from another overwrite. NULOAD also is used to load a partial load created on the PSS system.

5.2.2 Native-mode program change facility

To modify C-language programs in the system laboratory, an entire new process file (pfile) must be built on the PSS. Once this file has

been transferred via the LSS and TUS to the SPC 1B disk, it is installed on the 3B20D disk replacing the previous version of the process file. The next time the process is loaded into memory, the new pfile is used.

5.3 Laboratory simulators

Program testing in the TSPS system laboratory often requires the ability to either place a single call or a substantial traffic load on the TSPS and to automatically handle calls that require operator assistance in a live site. A number of microprocessor-controlled simulators including the Single-Line Simulator (SLS), Local Toll Simulator (LOTS), the Microprocessor-Controlled Load Box (MICLOB), and the Microprocessor Operator Position Simulator (MOPS) have been developed as part of the LSS to provide these capabilities for the TSPS No. 1 system and were also used to test the TSPS No. 1B. These call simulators are described in detail in Ref. 6.

VI. SUMMARY

The TSPS No. 1B Software Development System provides a complete set of facilities for TSPS generic development on the 3B20D Processor. Software generation tools can compile and link both emulated, assembly-language, and C-language programs. Administrative software and a rigorous development methodology control the evolution of feature development and keep track of the multiple versions of software that exist during the development process. Special-purpose software and hardware in the TSPS system laboratories provide a modern test environment for debugging and certification of the software product. These facilities are an effective and productive software development environment for future network operator services.

VII. ACKNOWLEDGMENTS

Many individuals have contributed in a significant way to the design and implementation of the TSPS No. 1B Software Development System. In particular, the authors wish to acknowledge the contributions of K. M. Conness, G. F. Gieraltowski, B. E. Holmes, M. S. Koehler, B. T. Rovegno, R. J. Welsch, and S. P. Winker, who were responsible for major components of the Software Development System. In addition, we would like to extend our thanks to D. L. Atkins, B. T. Rovegno, and E. S. Sachs for their many constructive suggestions during the preparation of this article.

REFERENCES

1. R. E. Staehler and J. I. Cochrane, "Traffic Service Position System No. 1B: Overview and Objectives," B.S.T.J., this issue.

2. R. J. Gill, G. J. Kujawinski, and E. H. Stredde, "Traffic Service Position System No. 1B: Real-Time Architecture Utilizing the DMERT Operating System," B.S.T.J., this issue.
3. B.S.T.J., *61*, No. 7, Part 3 (September 1982), special issue on the Stored Program Controlled Network.
4. D. M. Ritchie and K. Thompson, "*UNIX*™ Time-Sharing System: The *UNIX* Time-Sharing System," B.S.T.J., *57*, No. 6, Part 2 (July-August 1978), pp. 1905–30.
5. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "*UNIX*™ Time-Sharing System: The Programmer's Workbench," B.S.T.J., *57*, No. 6, Part 2 (July-August 1978), pp. 2177–2200.
6. J. J. Stanaway, Jr., J. J. Victor, and R. J. Welsch, "Software Development Tools," B.S.T.J., *58*, No. 6, Part 1 (July-August 1979), pp. 1307–34.
7. M. E. Barton, N. M. Haller, and G. W. Ricker, "Service Programs," B.S.T.J., *48*, No. 8 (October 1969), pp. 2866–80.
8. M. E. Barton, "The Macro Assembler, SWAP—A General Purpose Interpretive Processor," Fall Joint Computer Conference, 1970.
9. J. C. Lund, Jr., M. R. Ordun, and R. J. Wojcik, "Implementation of the Calling Card Service Capability-Application of a Software Methodology," Int. Commun. Conf., Denver, Colorado, June 1981.
10. B.S.T.J., *62*, No. 1, Part 2 (January 1983), special issue on the 3B20D Processor & DMERT Operating System.
11. G. T. Clark, H. A. Hilsinger, J. H. Tendick, and R. A. Weber, "Integration of the 3B20D Processor into TSPS," B.S.T.J., this issue.