Copyright © 1978 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 2, February 1978
Printed in U.S.A.

Common Channel Interoffice Signaling:

Development Tools

By J. S. COLSON, J. E. MASSERY, and G. A. RAACK

(Manuscript received May 7, 1977)

Tools used in the design, development, and testing of various components of the Common Channel Interoffice Signaling (CCIS) feature of No. 4A toll crossbar and No. 4 ESS are described. Included in the discussion are CCIS software design and administration support tools, electronic circuit design, analysis, and test tools, and laboratory support systems for software and hardware testing.

I. INTRODUCTION

A CCIS switching office or Signal Transfer Point (STP) consists of a variety of complex software and hardware systems whose sound design and thorough testing can be aided by effective support and development tools. Such software tools as a text editor, a macro assembler, and a linking loader are indispensable for program development. Managing large data bases of commonly used symbol definitions and large numbers of individual programs is accomplished by sophisticated software administration systems. Several hardware development tools not only aid the design and analysis of complex electronic circuits, but also provide circuit performance data used in the diagnostic software for these circuits. Support for the testing of completed software and hardware designs has also been provided. Laboratory utilities and test systems are available for debugging, function testing, and integration of system programs. The need for testing certain electronic peripheral circuits in an isolated environment has led to the development of off-line test systems that simulate the appropriate central control processors and their peripheral busses. The operation and application of each of these systems in the development of CCIS are detailed below.

II. SOFTWARE DEVELOPMENT TOOLS

The software development tools described below are programs and systems utilized in the design, development and maintenance of the CCIS real-time application software. Most of the tools are similar to those needed for any software development effort, and are representative of the major tools used during the development of CCIS. They are used for both the Stored Program Control (SPC) and Peripheral Bus Computer (PBC) developments and include a development and maintenance administration system, an editor, an assembler, a loader, and two special aids. One special aid is used for the development and maintenance of common pools of information, called COMPOOLS, which are used by the assembler. The other aid is a special purpose assembler for the CCIS terminal hardware unit to aid in assembling its language.

If we look at the development of a typical CCIS program module, called a pident, we can see how these tools are utilized. First, the name of the pident and its associated administrative information are entered into the Interactive Program Administration System (IPAS) data base. The user may then create the new pident through the use of an interactive editor within IPAS. The created pident, along with Advanced Processor Editor (APE) control cards, is submitted by IPAS for assembly by the appropriate version of the Switching Assembler Program (SWAP) as a batch job. Once the assembly is flag-free, the object module created by SWAP is linked to other modules of the system by the loader, which creates a load tape of the CCIS or PBC programs. This program tape is directly readable by the SPC or PBC machines, to initialize their memories with the real-time application programs.

For the CCIS development, these tools are designed to run on a general-purpose computer-center machine rather than on the application processor. The time-sharing and batch facilities of a large-scale, general-purpose computer are required to accommodate the heavy demand for these tools by the members of the development team; because the application processor is specially designed to control a switching machine, its instruction set and operating system are not suitable for general-purpose programming or time-shared use. Furthermore, to maximize the effective utilization of machine time, functions such as program editing and assembling, which can be carried out off line, are supported on the computer center processor, leaving the laboratory switching processor available for program debugging and system testing.

2.1 IPAS—the Interactive Program Administration System

IPAS is at the heart of the program development process. It executes interactively under the computer center's time-sharing facility. Before IPAS was available, programmers used card edit decks to define the changes required in a pident source file. Job Control Language (JCL) statements were generated by hand to run program assemblies under SWAP in a batch mode. IPAS replaced card edit decks with disk files of editor statements, and it replaced manual submission of assemblies with automatic generation and submission of required JCL. A programmer can now log onto IPAS, interactively create or modify a set of edit statements for a specified version of a pident, and schedule a batch assembly of the modified source without an intimate knowledge of the computer center operating system. Thus, more efficient use is made of program development time.

IPAS utilizes a central data base to control user access to IPAS, to control user access to particular functions and data files, and to record the existence and status of a pident's related edit files. The people in the Program Administration Group use the IPAS data base to control permanent changes to program source files, to check dates and times of assemblies, to generate official program listings for field distribution, and to set up loader input to produce a new generic tape.

2.2 Editing, assembling, and linking pidents

As with all Bell System stored program switching developments, such as No. 1 ESS, No. 2 ESS, and No. 3 ESS, an editor, assembler and loader are the basic tools for program system development. The APE editor used to edit the program source is that used by the No. 2 ESS and No. 3 ESS developments. The SWAP assembler used is similar to those used for other developments. The loaders which perform the linking functions for each machine are unique.

2.2.1 APE—the Advanced Processor Editor

A subject file data base is edited by the APE editor through the use of control cards and new input lines to produce a temporary updated file which is passed to the assembly step, and optionally, to a new, permanently altered and renumbered subject file. By using the APE editor, temporary changes can be made to the source files, and incorrect edits can be easily removed. In addition, the accumulation of these edits provides a history of changes to each pident, allowing programmers, testers, and administrators to determine the changes from one issue of a pident to the next very simply and efficiently.

2.2.2 SWAP—the Switching Assembler Program

SWAP is a powerful macro assembler, which reads symbolically coded machine instructions, pseudo-operations, and macros and converts them into object machine code.[1,2] Its normal outputs include an assembly listing and a disk data set containing the Object Program Module (OPM).

The OPM contains the assembled machine code plus linkage and administrative information needed by the loaders. Separate but similar versions of SWAP exist to assemble SPC and PBC code.

The SWAP assembler, developed concurrently with the SPC No. 1A processor[10] in the 1960s was intended for use by all electronic switching system software developments. In the next decade, when the use of high-level languages for SPC machines was investigated, the potential improvements in the program development process and in software maintainability did not appear to offset the penalty of greater real-time consumption and memory usage characteristic of most high-level languages. As a result, traditional methods using macros and assembly-language programming were employed in the development of 4A/CCIS software.

2.2.3 LDR—the Loader

Separate loaders exist for the SPC and PBC systems. Each loader takes any number of OPMs produced by SWAP (Section 2.2.2) and resolves the linkages between them. It also assigns each pident to a "real" piece of memory (address) or disk space and converts all relative addresses to absolute addresses. Each loader produces a listing consisting of the free and occupied areas of memory, the linkages resolved and outstanding, and any error messages. The ultimate product is the load tape, which is an application-machine-readable image of the linked real-time programs.

2.2.4 COMPAS—the COMPOOL Administration System

References to common data, formats, and locations are resolved not only in the loader stage, but also in the SWAP assembly stage using an entity known as a COMPOOL or DATAPOOL. COMPOOL is a collection of commonly used symbolic names, addresses, layouts (patterns) for locations, and registers, which are assembled and saved on a disk data set. A number of these preassembled disk data sets may then be used in subsequent SWAP assemblies to resolve references to symbols.

When used to refer to memory locations, the symbols contain certain special attributes, such as the address of a table, the address of a word or structure within a table, or a particular portion of a single word. To facilitate describing these symbols in a meaningful fashion, a special language was developed. This language, its compiler, editor, and data base are collectively known as the COMPOOL Administration System (COMPAS). The high-level language of COMPAS is used to describe the entities comprising a COMPOOL, such as tables, words, items, registers, constants, memory blocks, and holes in a hierarchical and sequentially

```
:SYSTEM         BUILD (CCIS)
:MOVEBEFORE     TABLEC, TABLEB
:INSERTAFTER    MYTABLE

    MEMBLK          addr1, addr2, PROTECTED, OFFICE_DATA,,,
                    'space for trunk tables and headcells'
    TABLE           TRKTABLE
    PROGRAMMER      'name room extension'
    SYSTEM          CCIS
    ORIGIN          ABSOLUTE, EVEN, 'used as list of headcells'
    MEMORY          PROTECTED, OFFICE_DATA
    LENGTH          16, 32, 'one word per trunk headcell'
    QUANTITY        1, 4, 'one table per trunk group — min=1, max=4'
    DOCUMENT        TTBL, 68009
    DESCRIPTION     'these tables are indexed by trunk group number'

        WORD            NAME=WORD1. TTBL, WRDTYP=DATA, WRDNUM=0
        PLACEMENT       'these headcells require immediate access'
        DESCRIPTION     'headcell is used to point to array of trunks'

            ITEM TTBL_EQUIP, 3, 17, N, 'eqpd=001, uneqpd=000, maint=100'
            ITEM TTBL_PTR, 17, 0, N, 'ptr to trunk tbl'

        WORD            NAME=TYPE_TTBL, WRDTYP=DATA, WRDNUM=1
        PLACEMENT       'corresponds to ttbl_ptr'
        DESCRIPTION     'used for trunk group type'

            ITEM TTBL_TYPE, 3, 0, N, 'see document PR-68003.I4 for
                 bit types'
            ITEM FILL, 17, 3,, 'unused bits', DEFAULT=0

    TCONST          TTBL_EQUIP_CHK, 1, 'constant used to check
                    equipped status'
    END_TABLE
    REGD            TTBL_REG_CHK, E (3), 0,, 'register definition used
                    to check trk tbl'
    HOLE            128, 'leave hole 128 words long in memory'

:RENAME         TRK_TBL, NEWTRK_TBL
:DELETE         HOLE. 17744
:PRINTAFTER     *ALL      #print formatted list of compool
:END
```

Fig. 1—Sample table definition and input commands for a COMPAS run.

ordered structure. Figure 1 shows a typical table definition in COMPAS format, with examples of the editing and layout commands.

The entities are entered into the data base, and the existing data base is manipulated by using the COMPAS command language. Thus, COMPAS provides the COMPOOL administrator with an entity-based editor and command structure. This feature facilitates the manipulation of complete entities such as tables, which have an arbitrarily complex structure and length, by using a single command.

COMPAS provides other special advantages over conventional COMPOOL defining techniques such as SWAP macros. It provides for the checking of the entity data for consistency. For instance, when the layout of a particular word is defined, the items can be checked to verify that all bits are defined once, unless declared otherwise. This provides a level of checking not possible using conventional SWAP declarations. COMPAS also provides for the complete description of each entity and its parts. This helps document the entities and provides for reference to the individuals responsible for controlling the entities.

In addition to the features already described, the COMPAS high-level language definition of a COMPOOL is transmitted to Western Electric, where it is used to build a data compiler automatically. This is done using a system developed jointly by Bell Laboratories and Western Electric known as the Integrated Data Management System (IDMS). This system facilitates the automatic updating of the data compiler needed to support changes to COMPOOL often required with issuance of new generics of the CCIS programs.

2.2.5 TASM—the Terminal Assembler

The terminal hardware unit,[3] which is a special purpose computer used as an interface between data transmission facilities and an application processor, is used in several switching systems. Currently, the terminal is able to communicate with the No. 1 ESS, No. 1A ESS, and SPC No. 1A processors. The terminal does not contain a peripheral unit such as a tape unit from which it is capable of loading its application program. Therefore, the terminal application program must be assembled by the respective SWAP assembler (Section 2.2.2) into the format of the application processor (SPC No. 1A, No. 1 ESS, or No. 1A ESS). That processor then can transmit the terminal application program to the terminal over its own communication paths.

The terminal has its own assembly language, so a terminal assembler (TASM) was written using SWAP macros and pseudo-operations. Thus, the terminal assembler is imbedded within SWAP in much the same way as was the CENTRAN (SNX360) assembler for the Safeguard project.[4] It is a one-pass data handling, two-pass program handling assembler.

The assembler consists of two parts: a common portion and an application portion. The common portion consists of approximately 1200 lines of macros and is used without change by all SWAP assemblers required to assemble terminal programs. The application portion is unique to each application (system) using a terminal. It consists of approximately 300 lines of macros which perform the job of packing the assembled data passed to it by the common section into the format necessary for the particular application.

Using this technique, the assembly listing produced contains the terminal source code lines, the assembled values and addresses in terminal format, and the packed application format, cross-referenced to the terminal format, in one listing. Also, as a result of this technique, a single SWAP assembly produces an OPM which can be linked by any one of the application-processor loaders.

III. HARDWARE DEVELOPMENT TOOLS

There were two software tools of major importance used in the development and testing of the hardware for CCIS. By far the largest and most complex tool was the Logic Analyzer for Maintenance Planning (LAMP).

3.1 LAMP—Logic Analyzer for Maintenance Planning

LAMP is a large and complex system which runs under several general-purpose computer operating systems. It is a circuit simulator capable of logic, fault, race, and timing analysis of circuits.[5,6] It was used to help design the CCIS circuits through provisioning for diagnostics and maintenance. It was also used to verify the logic and timing within the circuits prior to building laboratory models.

LAMP can produce outputs which link it to many other tools, such as the Diagnostic Language (DIAL) (Section 4.1) and the frame and circuit pack testing tools (Section VI). In particular, one of its outputs is used in the production of the Trouble Locating Manual (TLM), as described below.

3.2 TLM—the Trouble Location Manual Program

In order to locate and diagnose hardware problems in the new electronic circuits added for CCIS, a printed TLM using a first-failing test algorithm was provided. Production of such a TLM begins with one or more LAMP simulations of the circuit; one simulation may be run for each diagnostic phase. The input to each simulation is the data from the SPC-resident diagnostic programs. Results from multiple LAMP simulations are combined to form one "results" data base. These results, however, cannot be used directly to generate trouble numbers. Packing algorithms must first be applied to simulate the packing of results done within the SPC. The SPC diagnostic programs pack the results because of the limited SPC memory available for storage of the raw data. Different packing rules may be applied for each CCIS circuit.

A fault data base is constructed from the LAMP circuit model and from physical circuit data contained in circuit-pack device files. This data base associates the fault numbers used in the various processing algorithms (e.g., in LAMP), with physical locations and fault descriptions, and it defines the classes of equivalent (logically identical) faults.

The final step in TLM generation is the application of the trouble number calculation algorithm to the packed simulation results. The trouble number data is combined with the physical fault information to produce the printed behavioral TLM.

IV. COMBINATION HARDWARE-SOFTWARE DEVELOPMENT TOOL

DIAL is a macro language used to generate diagnostic tables for CCIS peripheral units. These tables are stored in SPC memory and in conjunction with a DIAL table executor, compose a diagnostic program. A typical DIAL statement may specify a peripheral order to the circuit under test and the corresponding expected reply to that order. The generation of the diagnostic tables is done using a DIAL-SPC compiler. The DIAL macros are also compiled using a DIAL-LAMP compiler to produce LAMP input vectors (Section 3.1). These input vectors are used to drive a LAMP simulation of the circuit to verify circuit operation during initial circuit design stages, to design and evaluate diagnostic tests, and to produce a TLM for the circuit through fault simulation. A third application of DIAL macros is to generate factory tests. The DIAL statements generate a data base which is released to Western Electric to be used to test the peripheral units before shipment.

Among the advantages of using the DIAL language are:

(*i*) The same set of source statements may be used during initial circuit design, in diagnostic generation, and in manufacturing test generation by inputting them to different DIAL compilers.

(*ii*) Functions are easier to code and understand because DIAL statements are macro calls.

(*iii*) The language is common to several peripheral units.

(*iv*) DIAL table-driven diagnostics require less SPC memory than machine-language code of the same tests.

In addition, because the DIAL compilers are actually a set of SWAP macros, functions coded in the DIAL language are portable and can be used in several machines and systems which use the SWAP assembler.

V. LABORATORY SUPPORT SYSTEM

The demand for increased reliability of software systems, coupled with the high degree of complexity which is characteristic of many modern software designs, has resulted in the need for effective and efficient testing methods and sophisticated laboratory support tools. The development of CCIS software for 4A crossbar and for No. 4 ESS—systems where the reliable performance of the software is essential to the continuity of telephone service—was supported by a number of such tools. (A discussion of No. 4 ESS support systems may be found in Ref. 7.)

5.1 Utilities for debugging and testing

When a program module is first introduced into the host processor in a laboratory environment, the software designer requires special tools which enable him to execute specific sections of his program, monitor its operation, detect and analyze performance anomalies, and rapidly make corrections and modifications. As system integration progresses, function testing causes increased program interaction, and additional testing aids are needed which provide for less disruptive collection of large amounts of performance data and rapid resolution of detected errors. Two independent laboratory utility systems, a host-processor-resident utility system and a minicomputer-resident noninteracting utility system, provide the program control and monitoring facilities required during the early stages of testing.

5.1.1 Resident Utility

Program testing at its most basic level is accomplished with the Resident Utility system, which is illustrated in Fig. 2. In this mode, the user is provided with the greatest degree of control over the execution of a program. By the use of the Noninteracting Utility Program Interface Console (NUPIC), which is used as a manual test console or "T-cart," program execution in the SPC processor may be stopped, instructions may be executed one at a time, or "matchers" may be used to detect the execution, reading, or writing of a specified memory location.

In addition to these manually controlled functions, the Resident Utility provides a variety of software-controlled features through a system of utility programs which "reside" in SPC memory. In either a batch mode using punched-card input, or interactively with teletypewriter (TTY) commands, the user is able to establish his test environment, control the execution of the program sections under test, and collect the desired run-time data. With the SPC system under the control of the utility system, the user may initialize internal registers and scratch memory, and cause execution to begin and end at given locations. By inserting special instructions at user-specified addresses, the utility system can monitor program progress at that address, dynamically modify run-time program parameters, or divert execution to special test routines. The transfer trace facility of the Resident Utility allows the printing of program addresses and internal registers each time a transfer instruction causes a break in sequential instruction execution.

A flexible program modification facility is an essential component of a laboratory utility system. The Resident Utility Overwrite Assembler is the means by which corrected program errors and modifications are incorporated into the machine-language version of the programs as they are being tested in the laboratory. Input statements to the Overwrite Assembler are compatible with the SWAP assembler (Section 2.2.2); once all additions and modifications have been tested in the laboratory to the satisfaction of the programmer, the overwrites may then be incorporated into the permanent version of the program using SWAP and its associated editing programs.

A variety of miscellaneous testing tools and laboratory aids are also part of the Resident Utility feature repertory. Memory may be dumped to magnetic tape or to the line printer, memory may be loaded from tape,

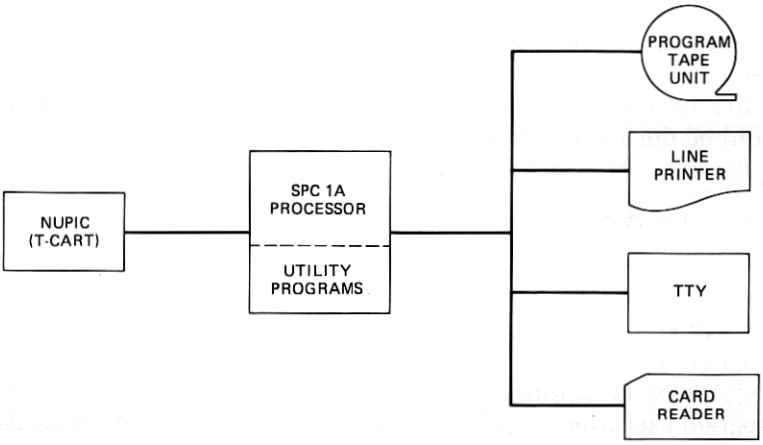

Fig. 2—Resident Utility system.

or data on a magnetic tape may be compared with that in memory. System software and hardware may be reinitialized in varying degrees under utility control.

However, a major drawback to the operation of the Resident Utility is the high degree of direct interaction between the utility and the programs under test. Execution of programs under test at their normal speed, without periodic interruption, is sacrificed for the fine control of program execution and the simplicity of the hardware structure characteristic of the Resident Utility.

5.1.2 Nonresident Utility

The ability to collect program execution and performance data without interaction by the utility system becomes essential in the program integration phase of testing. In a system such as 4A/CCIS, many call set-up functions are performed by electromechanical common-control hardware; the real-time software which controls and monitors this equipment executes essentially instantaneously relative to the much slower hardware. Interruptions to normal program flow, such as those caused by the Resident Utility to collect and print program data, could delay the initiation or execution of these programs, thereby distorting normal hardware-software sequences and corrupting test results. This inadequacy of the Resident Utility is overcome with the Nonresident Utility, whose noninterfering monitoring and off-line data processing are better suited to the more rigid environment of the latter phases of testing.

The nerve center of the Nonresident Utility system is a minicomputer

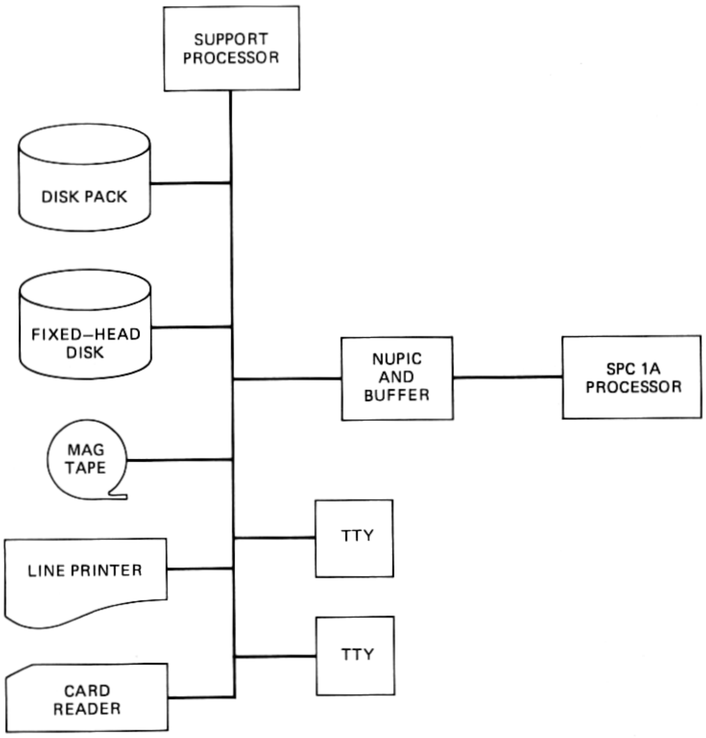

Fig. 3—Nonresident Utility system.

support processor (Fig. 3), one of whose peripherals is the NUPIC with its associated buffer. Under control of the Nonresident Utility software, the NUPIC is programmed to collect selected SPC program execution information when one or more of a variety of matchers detects a user-specified condition in the SPC processor. Among the conditions which these hardware matchers may be armed to detect are the execution of a specified SPC program address, the reading or writing of a given bit pattern at a specified address, and the accessing of a given SPC peripheral unit. In each case, the user may establish the particular conditions under which a match should occur either by composing Nonresident Utility statements interactively at a TTY keyboard or by identifying a previously created disk file containing images of the desired statements. Once these commands have been executed and the appropriate matchers armed, the NUPIC continuously monitors the operation of the SPC processor until a match occurs. At that instant, without interruption to the operation of the SPC, the NUPIC gates the information previously specified by the user's commands to a buffer which is unloaded by the support processor. An additional feature may be enabled or disabled when a match occurs:

noninteracting transfer trace, which provides a snap of critical SPC operational data each time a program transfer takes place. The autonomous matching and data collecting processes performed by the NUPIC allow for the gathering of a large amount of program execution data without disrupting the normal, full-speed operation of the system under test.

An important component of the Nonresident Utility system is the off-line data processing capability provided in the support processor software. While data being collected by the NUPIC is loaded into the hardware buffer, utility programs are unloading the data in its raw form onto a disk file. At the user's option, this data may be immediately translated into a readable form and printed at high speed, or may be stored on the disk for later off-line processing. A circular-file feature allows the automatic, continuous overlaying of the oldest collected data with new data. In this mode, the unneeded data from passed tests is automatically discarded, and only after a test failure or other irregularity is data collection stopped and the current file contents examined. The printed output of any data collection file may take a number of forms, including raw octal output, conversion to symbolic program names plus offsets, or printout only of data collected from a particular selected program.

Because of its rapid data handling and output capability, and its bulk storage facilities, the Nonresident Utility provides a number of other valuable tools and debugging aids. High-speed loading of SPC memory may be achieved either from magnetic tape or from a support processor disk file. SPC program and office data information may be rapidly dumped to tape, disk, or line printer. The noninterfering accumulation of large amounts of data, together with rapid and efficient off-line processing, have made the Nonresident Utility system an extremely effective testing tool.

5.2 4CAST—automated system testing

With the application of Common Channel Interoffice Signaling to the basic No. 4A toll crossbar system, the size and complexity of the software system has increased significantly. The architecture of the 4A/CCIS[8] and STP[9] machines, indeed, the structure of the entire signaling network, suggests that traditional testing techniques, while adequate for earlier switching systems, must yield to more flexible and powerful tools to keep pace with this advancing technology. The requirements for such a testing tool are that it be capable of communicating with a 4A/CCIS or STP machine over any of its various man-machine and machine-machine interfaces; that it be a convenient vehicle for the development, application, and administration of function and system tests; and that it provide sufficient flexibility and speed of operation to allow rapid execution of a large number of tests with a minimum of user intervention. These needs are met with the 4A/CCIS Automated Support and Test System (4CAST), an implement with which the testing of the large and complex CCIS software structure can be effectively managed. The 4CAST system consists of a compiler, which converts stimulus-response commands written in a high-level language into a command-table load module, and a laboratory run-time system, which executes the load module commands.

5.2.1 Compiler

The 4CAST language, consisting of keywords and structures similar to those in PL/1, enables the test designer to convert test specifications into sequences of action directives or response monitors in a form that is easy to generate and understand. A single 4CAST "procedure," or compilation entity, is typically produced for each test definition and compiled by the 4CAST compiler, which runs on a general-purpose computer center processor. As shown in Fig. 4, the generation of a procedure begins with the user's coding of the procedure text in the 4CAST

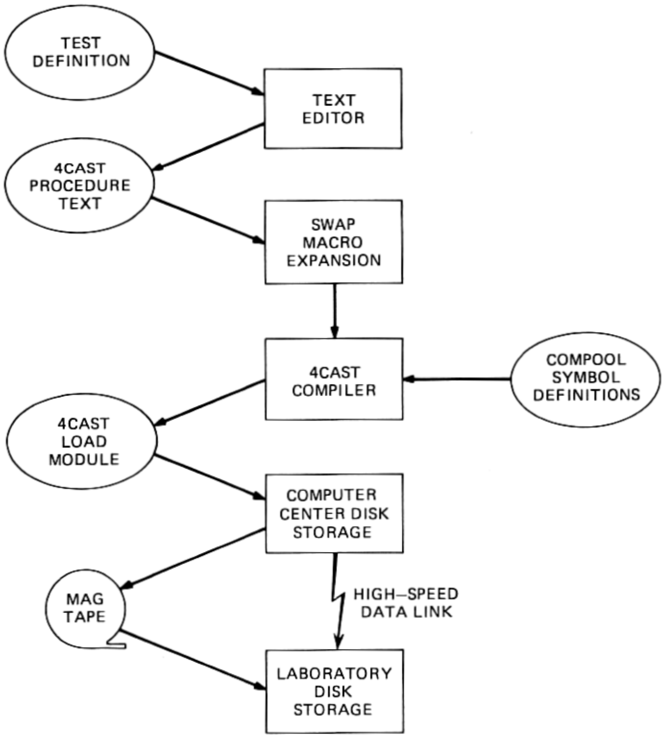

Fig. 4—Generation of 4CAST procedures.

```
PROCEDURE    TESTX. CONTROL;

DECLARE
   INTEGER     X = 25, Y = B (110100);
   INTEGER     SPC_ADDR;                   # definition in COMPOOL;
   TEXT        ERR1 = "INVALID DIGITS";
   TIMER       T1 = S (15), T2 = MS (200);
   SU          ANSWER, HANGUP;             # supervisory signals;
   IAM         ADDR_DIGITS = C (5551212);  # telephone number;
                     .
                     .
                     .

END;

START:
   RUN INIT (PARAM1, PARAM2);                 # initialize;
   DELAY T2;
   SENDSU  ADDR_DIGITS . trunk_number;        # initiate call;
                     .
                     .
                     .

   ON  ANSWER . trunk_number
      DO;
         PRINT "CALL ANSWERED";
         IF READSPC (SPC_ADDR) = Y
            THEN GO TO RESTART;
            ELSE CONTINUE;
      END;
   WAIT;
                     .
                     .
                     .

RESTART:
   SENDSU  HANGUP . trunk_number;             # disconnect call;
   PARAM1 = PARAM1 + X;
   GO TO START;
END TESTX;
```

Fig. 5—Sample 4CAST test.

language. For added flexibility and convenience, user-defined macros for repetitive or complex functions may be expanded by the SWAP assembler's macro facility. The 4CAST compiler then converts the text commands, definitions, and directives into a 4CAST load module containing tables which drive the laboratory Run-Time System. The compilation process also provides access to the common pool (COMPOOL) of symbol definitions used in the assembly of SPC and CCIS programs. Figure 5 is a sample of the text of a simple 4CAST procedure. Once compiled, a procedure's load module is transported to the laboratory either on a magnetic tape or directly from a computer center disk file to the laboratory support processor disk via a high-speed intermachine data link.

5.2.2 Run-Time System

Once the 4CAST load modules have been transferred to the 4CAST processor's disk, they may be executed by the 4CAST Run-Time System. In the unattended mode of operation, a list of "master procedure" names is entered by keyboard command to the Run-Time System. Each master procedure contains 4CAST directives which load, start, and exit individual tests, or "control procedures," each of which is a 4CAST load

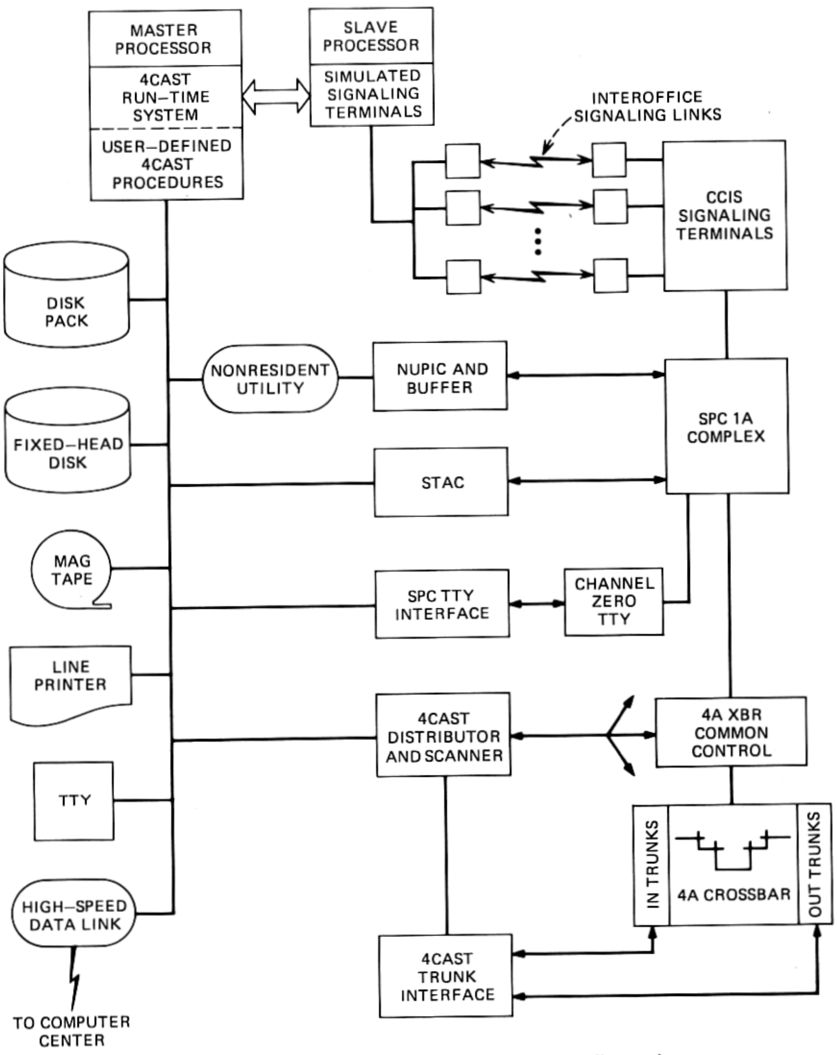

Fig. 6—4CAST system in switching office configuration.

module which performs the initialization, defines the required actions, and monitors the results of a given test. Control procedures are capable of initiating "subprocedures" to perform basic, repeated tasks, such as the set-up of a particular type of call, or the removal from service of a given signaling link. Communication between procedures is accomplished either with parameters passed from a calling procedure (master or control) to a subordinate procedure, or through a common scratch area accessible by all executing procedures.

As seen in Fig. 6, 4CAST can access the CCIS switching office or STP systems in a variety of ways. The most fundamental communication channel is through the CCIS signaling links from the 4CAST Simulated Terminal Interface. It is through this link that nearly all interoffice signaling is simulated. CCIS call set-up and disconnect, signaling network maintenance and control, and other interoffice communication can be initiated and monitored automatically by sequences of user-specified commands in 4CAST procedures. Conventional call origination and termination is accomplished through the 4CAST Trunk Interface, which controls signaling to a variety of conventional 4A trunks. The Distributor/Scanner Circuit, in addition to driving the Trunk Interface, provides direct access to and control over functions in the 4A hardware and in the SPC complex. The primary man-machine interface, the SPC maintenance TTY, is controllable through the TTY Interface using a number of text-handling commands and options in the 4CAST language. Using run-time processing of text variables, the user's procedure may simulate a TTY dialogue with the SPC. Under special circumstances, a test sequence may require that the 4CAST procedure have access to SPC memory or to internal SPC processes. For this reason, two channels are provided which allow direct interaction between 4CAST and the SPC processor: the Nonresident Utility interface using the NUPIC, for performing such utility functions as setting matchers, and the Simulated Terminal Access Circuit (STAC), which allows 4CAST to momentarily halt the SPC, allowing the gathering of internal status or progress data, or the reinitialization of large blocks of memory. Additional features of the 4CAST language and Run-Time System provide for arithmetic functions, command execution control, timing facilities, and data gathering and handling control.

Because of the programmable nature of 4CAST test procedures, changes may be made quickly and easily, allowing test designers to keep pace with the often rapid evolution of the software system under test. An easily manageable administration system for 4CAST tests permits the reapplication of all, or certain subsets, of the existing tests to subsequent issues or generics of the software. For function testing, system integration testing, and regression testing of CCIS software, 4CAST provides an important facility for the generation and application of tests.

VI. OFF-LINE HARDWARE TEST TOOLS

Three categories of testing necessitated the development of off-line test systems for the new electronic peripherals developed for CCIS. These categories are:

(*i*) Laboratory testing of prototype hardware by the circuit designer.

(*ii*) Preliminary testing of diagnostic software against prototype hardware.

(*iii*) Manufacturing testing of standard production hardware.

The peripherals for which this capability was developed are the No. 4A, No. 4 ESS, and No. 1 ESS Terminal Groups, the No. 4A Distributor and Scanner, and the No. 4A/No. 4 ESS and No. 1 ESS terminal units.[3] With the exception of the latter two units, these peripherals share a common characteristic—they are controlled by commands from stored program processors via well-defined bus structures. This characteristic, plus the need for interactive testing and access to large, computer-generated data bases, indicated a computer-controlled system with input media compatible with the LAMP-generated data bases and output interfaces that simulate either the SPC No. 1A,[10] the No. 1A ESS,[11] or the No. 1 ESS[12] processor peripheral bus structures.

In general, each test system is configured as shown in Fig. 7. A minicomputer controls the application of tests to the peripheral under test and compares the peripheral's response with the expected response. The bus interface, or simulator, generates signals of the level and duration defined for the processor bus structure being simulated. Manual control is provided to allow the test engineer to generate special tests which may not exist in the computer-generated test file.

Test files for each peripheral are typically derived from the LAMP simulation data base created during the development of diagnostic programs. As a consequence, the diagnostic information is subjected to an early test against prototype hardware. Once they are generated and resident in the minicomputer, a test monitor program allows access to single tests, groups of tests, or phases, allows repetitive application of a single test or phase in a loop, and provides for on-line editing of the test

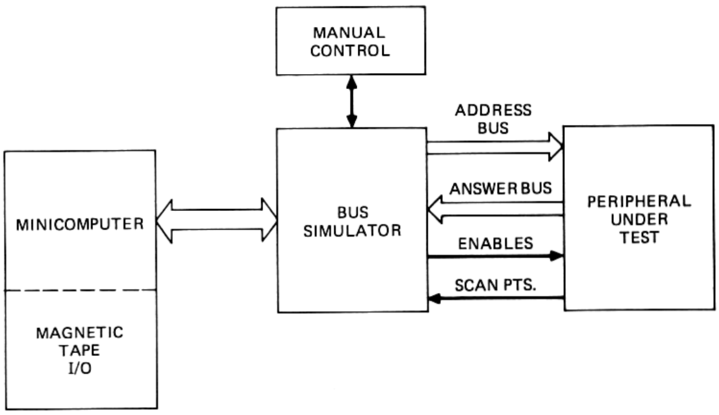

Fig. 7—Test system general configuration.

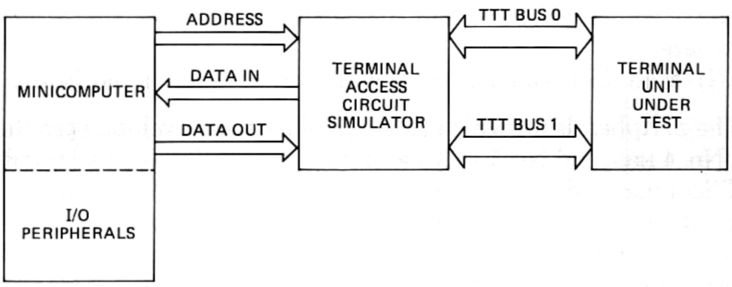

Fig. 8—Terminal unit test system configuration.

file. Tests for circuits not simulated or tests written for the manufacturing testing environment may be run under control of the monitor in conjunction with the computer-generated file.

In the case of the terminal unit test systems, rather than the processor bus structure being simulated, the Terminal Access Controller (CONT)[3] or Terminal Access Circuit (TAC)[3] interface is modeled. Figure 8 illustrates the general test system configuration for these units. The simulation data bases in this case are translated into TAC- or CONT-to-terminal operation codes and data, and the terminal responses are compared by the minicomputer with those predicted by the simulation.

The off-line test system hardware and software designs also provide flexible and efficient tools with which Western Electric is able to conduct manufacturing tests on the respective peripheral frames.

Off-line facilities are also provided for the test and evaluation of individual circuit pack designs, and for testing frames and units at extremes of temperature and humidity.

VII. CONCLUSION

From early design analysis through system integration and manufacturing tests, support tools provided an environment for the efficient and productive development of each of the components comprising the toll CCIS switching and signaling systems. As the development of new features is undertaken to exploit the flexibility and potential of the CCIS network, support systems will be relied upon more heavily to assist in the administration and testing of new designs. Advancing technology in the field of hardware and software support systems, as well as experience gained during initial CCIS development, will enable us to keep pace with this demand for increased development support capability.

REFERENCES

1. M. E. Barton, N. M. Haller, and G. W. Ricker, "Service Programs," B.S.T.J., *48,* No. 8 (October 1969), pp. 2865–2896.
2. M. E. Barton, "The Macro Assembler, SWAP—A General Purpose Interpretive Processor," Proc. AFIPS F.J.C.C., *37* (1970), pp. 1–8.

3. B. Kaskey, et al., "CCIS: Technology and Hardware," B.S.T.J., this issue.
4. B. N. Dickman, "CENTRAN—A Case History in Extendible Language Design," B.S.T.J., Safeguard Special Supplement, pp. S161–S172.
5. H. Y. Chang, G. W. Smith, Jr., and R. B. Walford, "LAMP: System Description," B.S.T.J., *53,* No. 8 (October 1974), pp. 1431–1449.
6. T. T. Butler, T. G. Hallin, J. J. Kulzer, and K. W. Johnson, "LAMP: Application to Switching System Development," B.S.T.J., *53,* No. 8 (October 1974), pp. 1535–1555.
7. P. S. McCabe, J. B. Otto, S. Roy, G. A. Sellers, Jr., and K. W. Zweifel, "No. 4 ESS: Program Administration, Test and Evaluation," B.S.T.J., *56,* No. 7 (September 1977), pp. 1239–1278.
8. K. E. Crawford, C. J. Funk, P. R. Miller, J. D. Sipes, and R. C. Snare, "CCIS: 4A Toll Crossbar Application," B.S.T.J., this issue.
9. P. R. Miller, R. C. Snare, and R. E. Wallace, "CCIS: Signaling Network," B.S.T.J., this issue.
10. G. R. Durney, H. W. Kettler, E. M. Prell, G. Riddell, and W. B. Rohn, "TSPS No. 1: Stored Program Control No. 1A," B.S.T.J., *49,* No. 10 (December 1970), pp. 2445–2508.
11. 1A Processor Special Issue, B.S.T.J., *56,* No. 2 (February 1977).
12. J. A. Harr, F. F. Taylor, and W. Ulrich, "Organization of No. 1 ESS Central Processor," B.S.T.J., *43,* No. 5 (September 1964), Part 1, pp. 1845–1922.