INTRODUCTION

The SYMBOL system is a full-scale working demonstration of an ALGOL-like, high-level, general-purpose, procedural language and a multi-processing, multi-programming operating system implemented directly in hardware. The results have proven the versatility of the design, construction, and testing techniques that were developed to enable a small number of people to implement such a major system in a reasonable time and at a relatively low cost.

Although the last items in the project to be developed, the methods used to test and debug SYMBOL were not the least important; delivery of the finished system would have been much delayed if more conventional methods of manual excitation and probing had been used. To facilitate fuller understanding of these testing techniques, a brief description of the architecture and physical characteristics of SYMBOL will be given first, followed by a more detailed description of the test facilities.

SYMBOL HARDWARE AND ARCHITECTURE

Logic in SYMBOL is implemented with Complementary Transistor Micro-Logic (CTµL), a positive AND/OR logic family with wired-OR capability at every output except that of the flip-flop. The importance of the wired-OR capability must be emphasized: there are a significant number of two-way wired-OR ties with sources throughout the six-foot long main frame; if discrete OR gates had been required, the size of the finished system would have been considerably larger and its simple bus-oriented organization would not have been possible.

The dual-inline CTµL integrated circuits are assembled on about 110 large two-sided printed circuit boards with plated-through holes (see Figure 1). These boards contain all of the logic components in the system and are easily removed for incorporation of logic changes using discrete wire “patches” to the original printed circuitry. Figure 2 illustrates schematically how the boards are mounted between a pair of parallel two-sided printed circuit “motherboards”.

Communication between boards is provided by two groups of 200 parallel printed circuit bus lines on the motherboards, the “connect” bus and the “bypass” bus. These buses provide all required system interconnections except for cables to peripheral devices and to the console. Physically, each large board can contact up to 200 bus lines and have the remaining 200 lines bypass it. A signal on the bypass bus may cross over a signal on the connect bus between board positions to allow a board to contact the corresponding position on the adjacent board or, alternatively, to a board several slots distant. In addition, either or both buses may have continuity breaks to isolate groups of boards.

There were eleven such isolated groups of boards in the original system, where each such Autonomous Functional Unit (AFU) acts as a specialized processor (see Figure 3). The Main Bus, a 111-wire subset of the 400 wires on the motherboards, runs the full length of the system and connects to each AFU. The lines in this time-shared, global communication path are distributed as follows:

- 64 bidirectional data lines
- 24 bidirectional address lines
- 10 bidirectional priority lines
- 6 operation code lines
- 5 terminal number lines
- 1 system clock
- 1 system reset

In order to fully appreciate the operation of the debugging facilities and the interactions between these facilities and SYMBOL, it is necessary first to under-
Figure 1—Typical 12" × 17" two-sided printed circuit board with logic additions and changes made with discrete wires

stand the various modes of communication which exist between the eleven AFU’s. The foundation for this understanding is the knowledge that everything of importance that happens within an AFU eventually has some effect on the Main Bus. When the Instruction Sequencer finishes processing one instruction and needs another, a request to the Memory Controller is sent via the Main Bus. If the Arithmetic Processor runs out of data, a call for more goes to the Memory Controller. In both cases, information returned from the Memory Controller is sent over the Main Bus. If an input device completes its operation, the Channel Controller notifies the Interface Processor via the Main Bus. When the Translator is done with compilation of a program, the System Supervisor is notified via the Main Bus. Even if a processor fails catastrophically and refuses to function at all, the absence of Main Bus transfers originating from that AFU over a short period of time (20 clock cycles, perhaps) can be detected. Thus it can be said that whatever controls or watches the Main Bus thereby controls or watches SYMBOL. This concept of the Main Bus and the kinds of information that are transmitted through it are key features in the design of the total system and in the functioning of the debugging facilities.

Four distinct types of bus usage are possible. They are:

AFU to Memory Controller requests
Memory Controller to AFU returns
AFU to AFU transfers
Control exchange cycles

All Main Bus transfers are priority sequenced, where the priority bus coordinates the usage of the other buses. When an AFU desires to use the Main Bus, it raises its priority line and simultaneously checks to see if there are any higher priority requests pending; if there are none, it uses the bus on the following cycle. AFU to AFU cycles, used only between sections of the Central Processor, are also controlled by the priority bus: the Central Processor assumes the Main Bus is available for a private “sneak” cycle if no priority lines are raised. Control exchange cycles are used to communicate control information between the System Supervisor and the other AFU’s. During a control cycle, the Main Bus lines have preassigned uses, with certain lines used to start and stop each AFU, others to indicate the completion of an AFU task, etc.; during a given
cycle, any combination of the paths can be used simultaneously.

SYMBOL DEBUGGING

The testing operations required in the development of a large system can be divided naturally into several fairly specific stages. The test functions vary as the system progresses from the research and development stage, to the prototype stage, to the production stage, and finally to the customer support stage. The emphasis on testing during these stages shifts from one in which strong support is given to engineering design checkout, to one which gives strong support to bad-part identification. System development within each stage requires component, board, AFU, and system testing functions; these functions are not always clearly distinct, but they can be considered separately for convenience. The discussion which follows illustrates the major requirements for testing in a research and development environment, with other aspects of testing discussed to a lesser extent.

Component testing

Component testing is normally an incoming inspection operation. The integrated circuit devices used in SYMBOL were checked both statically and dynamically upon receipt for two main reasons: (1) The devices often had a long lead time for procurement, and an incoming inspection allowed immediate replacement orders to be made for bad devices. (2) The environment in which the devices were used (printed circuit boards) is one which has a high time (and therefore cost) penalty for replacing a device that should not have been installed in the first place.

The tests included a static check of the logic the device was supposed to implement, the on/off delay times from threshold to threshold, a check on the ringing tendencies, operation of the device when driven by a minimum clock pulse (where applicable), and the operation of the device under minimum and maximum power supply limits. These tests were made on a breadboarded device tester which, while adequate, was not very efficient, especially from a human factors viewpoint. In particular, the tester was capable of making only one class of measurements at a time; if a device required two classes of measurements, it had to be retested in a second pass. This resulted in a low throughput rate since about half of the time required to test a device was spent in manually transferring it from the source bin to the tester and from the tester to the “accept” or “reject” bin. Fortunately, better component testers are now available with good human factors design and automatic device transporters for quick and efficient incoming inspection.

Except for perhaps 100 devices received in the last few weeks of the project, all 18,000 integrated circuits in SYMBOL were given a thorough incoming inspection. It is safe to say that these last 100 devices caused more trouble during debugging than the first 18,000. This was sufficient proof of the absolute necessity for complete incoming component testing for future projects.

Board testing

Two major kinds of errors should be detected and corrected through board testing before a board is placed into a system. As in most system development, construction errors of various kinds will be discovered.

Figure 3—Block diagram of the SYMBOL system showing the original eleven autonomous functional units all connecting to the main bus.
These errors will include fabrication errors such as drawing mistakes, missing or misplaced holes in the printed circuit, solder bridges, and misplaced components, to name just a few. In addition, there will also be system design errors ("oversights") which must be found and corrected. The scope of this problem was extremely large for SYMBOL boards because of their great number (110), their size (maximum of 220 integrated circuits), and their limited number of signal contact points (250). Considerable development is still necessary before automatic computer test generation is economically feasible for this size of board; current programs might require two hours to generate complete tests for about one-fifth of the logic on a board, and the difficulty of the task increases considerably with increasing size. Such costs could not be justified for this prototype system. If the problem were altered from an exhaustive test of all of the logic on a board to a test of all the used states of the logic, it might very well prove economically feasible. In particular, in a production environment, the cost of such test generation could be amortized over many systems since the same test sequence would be used over and over again. However, even though the isolation and correction of fabrication errors before system checkout began would have allowed more efficient debugging, it was felt that the amount of effort required for a thorough precheck could not be justified for this one-of-a-kind system. Thus, after a relatively cursory examination of each board for obvious errors (such as solder bridges) by experienced technical help, the board was returned to the designer for simple logic verification in a manual test station before proceeding to AFU or system test. This board test station has been described elsewhere.

Subsystem and system test

Subsystem testing was the process of debugging a complete Autonomous Functional Unit as a unit. Each AFU performs, by itself, some function in the complete system, and, in a sense, is a stand-alone unit. Except for the Memory Reclaimer, a two-board AFU which was extensively (but not completely, as it turned out) simulated using the FAIRSIM* system, most AFU’s were too large to allow simulation as a unit; therefore, subsystem test was the first time at which all of the boards in an AFU were run together. By this time most of the fabrication errors had been detected through board testing, although there were always a few left to find.

Subsystem testing could take many forms and could be performed in many different ways. An extension of the board test station was conceived where all boards of an AFU could be inserted into one module and a manual test performed. This sort of testing might have been useful in some limited areas such as verifying the basic cycling of control logic or the basic register transfers within an AFU, but, in general, a complete AFU test in this sort of test station would be very difficult, inefficient, and time consuming. Recognizing, then, that a thorough job of AFU testing was not desirable or even practical, subsystem and system testing were combined into the same operation and were essentially carried on simultaneously.

System testing was quite a prolonged process and involved a wide spectrum of activities from the initial startup of each of the AFU’s on through to the final operation of the full system. The debugging of SYMBOL encountered many unique problems analogous to trying to test both a central processing unit and an operating system simultaneously on a more conventional computer. Unfortunately, self-diagnostics were impractical until late in the project because the construction schedule was such that Input/Output equipment was one of the last items to become operational. Some way of automatically exercising and/or monitoring various AFU’s and portions of the system and of recording the results of these tests was absolutely necessary; in response to these needs, the first System Tester was developed.

SYSTEM TESTER

The System Tester consisted of three major elements: a programmable minicomputer, a logic interface and bus multiplexer between the minicomputer and SYMBOL, and a comprehensive set of computer programs which could control (and be controlled by) events

* Fairchild Digital Simulation program for Computer Aided Design.
occurring on the Main Bus. The configuration is schematically represented in Figure 4.

The minicomputer was an IBM 1130 with an 8192 word, 16 bits per word, memory (3.6 microsecond cycle time), a removable disc, a console keyboard and typewriter, a card reader/punch, a Data Products 80-column high-speed line printer, and a Storage Access Channel. The standard software supplied with the system (Disk Monitor Version II, Level 3) was used throughout the project.

The hardware interface consisted of four large logic boards between a pair of short motherboards mounted in a small separate enclosure (along with the necessary power supplies, cooling fans, etc.), placed adjacent to one end of the SYMBOL main frame. Short cables were used to connect the Main Bus to the interface, and a somewhat longer and thinner cable connected the interface to the control computer. The interface to the line printer was located on a fifth board in this same module.

The System Tester interface logic was divided into four basic sections. The first was a fairly small one which consisted of a number of TTPuL inverters which were the logic level buffers between the control computer and the CTpL logic in the rest of the system. Second was a set of control interface registers and Input/Output logic which communicated with the control computer. The third section was another set of data registers which could be loaded and unloaded by commands from the control computer and which were used to transfer data to and from the Main Bus. The last section was the control logic which obeyed the priority rules of the Main Bus and included system reset and clock controls.

In addition to the 111 Main Bus lines, four extra lines were allocated to a System Test Mode which was distributed throughout the system and decoded by certain AFU's to provide additional control of several of the more complex operations. Different test modes, for example, could completely turn off a particular AFU or could force serial instead of parallel operation of all the AFU's. The discrepancy between the speed of the control computer's 16-bit data channel and the considerably larger Main Bus was resolved by allowing the interface to turn off the SYMBOL clock momentarily each time data was transferred to or from the interface buffer registers. Although control of the clock in this manner significantly lowered the execution speed of SYMBOL, all logic sequences were still executed in the same order with respect to each other, except for real-time peripheral devices, no matter what the actual clock speed.

The third major element of the System Tester was a language called DEBUG (what else!) and a set of programs which interpretively executed this language to control the interface as directed by input from the user; details of the language are given below. The actual interface and interrupt-serving subroutines were written in assembly language while the majority of the remaining programs were written in FORTRAN. However, as new features were added to the system during the development period, the limited memory of the control computer was exhausted several times; each time, another routine was converted from FORTRAN to assembly language in order to free additional space. Notwithstanding this severe memory limitation, only rarely was a language feature removed, and then only after assurances from all users that the feature really was not being used.

In addition to the specific language features described below, many specialized programs were written (normally in FORTRAN) to provide formatted memory dumps, save-restore capability between the removable disc and SYMBOL memory, and other convenient features.

**DEBUG LANGUAGE**

The following sections describe the DEBUG language and the operation of the System Tester in some detail and especially attempt to convey some of the flavor of the system. The past tense is used because the System Tester was replaced by a special-purpose AFU, the Maintenance Unit (to be described later) and thus no longer exists.

The interpretive DEBUG processor accepted commands from punched cards. Each command consisted of an instruction element followed by a list of zero or more data elements. The format was relatively free form, with all blanks (spaces) ignored except in a string data element. Each element was separated from the preceding one by a comma or an equal sign. The last element was terminated by a period. Comments or other information following the period were ignored (a comment card was one whose first non-blank character was a period). Blank cards could be included in the deck for readability.

Extensive error checking was performed by the DEBUG processor, although very little error recovery was attempted because of the paucity of memory. When a mistake was discovered, the main response was an indication on the output listing of its nature. The system's action after an error was not predicted. In addition to a listing of the input commands and error messages, other messages of an informative nature and
TABLE I—List and Description of Programmed Register and Interface Hardware Register Mnemonics Recognized by the DEBUG Processor

<table>
<thead>
<tr>
<th>Programmed Register Mnemonics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMB</td>
<td>Address last formed by a COMBINE command</td>
</tr>
<tr>
<td>COUNT</td>
<td>Number of Main Bus transfers monitored in tracing</td>
</tr>
<tr>
<td>SPLTG</td>
<td>Group-link word address from a SPLIT command</td>
</tr>
<tr>
<td>SPLTP</td>
<td>Page address from a SPLIT command</td>
</tr>
<tr>
<td>SPLTW</td>
<td>Word address from a SPLIT command</td>
</tr>
<tr>
<td>ZERO</td>
<td>Infinite source of zeros</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface Hardware Register Mnemonics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP</td>
<td>Monitored SYMBOL priority bus lines</td>
</tr>
<tr>
<td>OP</td>
<td>Memory operation code last transmitted or error code last received</td>
</tr>
<tr>
<td>PL</td>
<td>Page-list currently specified (the mnemonic PAGE LIST could also be used)</td>
</tr>
<tr>
<td>PRIOR</td>
<td>Priority of the System Tester (any of the AFU's could be simulated)</td>
</tr>
<tr>
<td>RADR</td>
<td>Address last returned from SYMBOL</td>
</tr>
<tr>
<td>RCVX</td>
<td>Patchable register last received from SYMBOL</td>
</tr>
<tr>
<td>RDATA</td>
<td>Data last returned from SYMBOL</td>
</tr>
<tr>
<td>TN</td>
<td>Terminal number currently in use (TERMINAL NUMBER could also be used)</td>
</tr>
<tr>
<td>XADR</td>
<td>Address last transmitted to SYMBOL</td>
</tr>
<tr>
<td>XDATA</td>
<td>Data last transmitted to SYMBOL</td>
</tr>
<tr>
<td>XMTX</td>
<td>Patchable register last transmitted</td>
</tr>
</tbody>
</table>

the results of any action taken could be obtained on the line printer.

Data elements

A data element could have one of the following forms:

1. A hexadecimal constant formed by the character "/" followed by zero or more hexadecimal digits. Only the last 16 were retained. A typical example is /0079 004F CO 02D63B, where blanks have been included for readability. If fewer than 16 digits were entered, the result was right-justified. For left-justification ("contraction" is a good mnemonic) the last digit could be followed by an apostrophe and a character to designate the "fill" digit. Examples are /FF'O, where the leftmost byte is all ones and the rest are zeros, and /'A, a word of alternating ones and zeros.

2. A positive decimal number formed by zero or more decimal digits. Two examples are 0 (zero) and 18 446 744 073 709 551 615, the largest possible 64-bit number.

3. An alphanumeric string formed by a string-start indicator (an apostrophe) followed by zero to eight characters. To allow for character codes not represented on a keypunch, triplets composed of an escape character (the @) followed by a two-hex-digit coding of the desired character could be included in the alphanumeric string.

4. A data element could also be a register mnemonic formed by a single letter "A" through "Z" or a word chosen from a list of programmed registers or interface hardware registers which had specific predefined uses. The single-letter programmed registers were for temporary storage of data, addresses, etc., as the user desired. They could be initialized, cleared, etc., only by DEBUG commands or by being explicitly named in a Memory Control or System Supervisor command. A list and description of the multi-letter register mnemonics is given in Table I. Programmed registers A through Z and ZERO and hardware registers XDATA and RDATA were eight-byte (64 bit) registers. To refer to fewer than eight bytes, the mnemonics could be followed by a subscript from 0 through 7 enclosed in parentheses. Subscripts 3, 4, 6, and 7 implicitly referenced a single-byte datum. Subscripts 0 and 2 referred to a two-byte page address, and subscripts 1 and 5 referenced three-byte word addresses. No subscript implied context-dependent use of some or all of the full 64-bit register.

5. And finally, a data element could be empty or null if it were formed by two consecutive terminators or two terminators separated only by blanks.

The System Tester was a combination hardware/software system. To execute each command, the DEBUG processor transferred the contents of the indicated programmed registers to the appropriate hardware registers in the interface and initiated the proper operation. Although a data element could be left unspecified, the corresponding hardware interface register still participated in the operation; its contents were simply the value remaining from the previous operation. These registers could be initialized, set, cleared, etc., by DEBUG commands or by implicit side effects of various Memory Controller or System Supervisor operations.

Instruction elements

An instruction element was a mnemonic chosen from a predefined list; only the first five characters of mnemonics containing more than five characters were used...
TABLE II—List of All DEBUG Main Bus Transfer Commands with Data Elements Explicitly Identified by the Corresponding Default Hardware Interface Register Mnemonic

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
<th>Side Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG, XADR, RADR.</td>
<td>Assign group</td>
<td>RDATA = 0</td>
</tr>
<tr>
<td>AG00, RADR.</td>
<td></td>
<td>XADR = RDATA = 0 = PL</td>
</tr>
<tr>
<td>AG01, RADR.</td>
<td></td>
<td>XADR = RDATA = 0</td>
</tr>
<tr>
<td>AG10, XADR, RADR.</td>
<td></td>
<td>RDATA = 0 = PL</td>
</tr>
<tr>
<td>DE, XADR.</td>
<td>Delete end of group</td>
<td>RDATA = RDATA = 0</td>
</tr>
<tr>
<td>DS, XADR.</td>
<td>Delete string</td>
<td>RDATA = RDATA = 0</td>
</tr>
<tr>
<td>DL, XADR.</td>
<td>Delete page list</td>
<td>RDATA = RDATA = 0</td>
</tr>
<tr>
<td>FD, XADR, RADR, RDATA.</td>
<td>Fetch direct</td>
<td>RDATA = RDATA = 0</td>
</tr>
<tr>
<td>FF, XADR, RADR, RDATA.</td>
<td>Fetch and follow</td>
<td>RDATA = RDATA = 0</td>
</tr>
<tr>
<td>FL, XADR, RADR, RDATA.</td>
<td>Follow and fetch</td>
<td>RDATA = RDATA = 0</td>
</tr>
<tr>
<td>FR, XADR, RADR, RDATA.</td>
<td>Reverse follow and fetch</td>
<td>RDATA = RDATA = 0</td>
</tr>
<tr>
<td>IG, XADR, RADR.</td>
<td>Insert Group</td>
<td>RDATA = 0</td>
</tr>
<tr>
<td>RG, XADR, RDATA.</td>
<td>Reclaim group</td>
<td>RDATA = RDATA = 0</td>
</tr>
<tr>
<td>RG0, RDATA.</td>
<td></td>
<td>XADR = RDATA = 0</td>
</tr>
<tr>
<td>SA, XADR, RADR, XDATA.</td>
<td>Store and assign</td>
<td>RDATA = 0</td>
</tr>
<tr>
<td>SD, XADR, RADR, XDATA.</td>
<td>Store direct</td>
<td>RDATA = 0</td>
</tr>
<tr>
<td>SI, XADR, RADR, XDATA.</td>
<td>Store and insert</td>
<td>RDATA = 0</td>
</tr>
<tr>
<td>SO, XADR, RADR, XDATA.</td>
<td>Store only</td>
<td>RDATA = 0</td>
</tr>
<tr>
<td>SS, XDATA, XADR, OP, RDATA.</td>
<td>Control exchange cycle</td>
<td>RDATA = 0</td>
</tr>
</tbody>
</table>

An example of each Main Bus command, with data elements explicitly identified by the default hardware interface register mnemonic, is given in Table II.

In addition to the simple Main Bus commands, there were 38 other commands available to the user to provide for explicit control of the System Tester environment; control of the hardware interface registers; maintenance of the files stored on the control computer's disc; loading, reading, writing, and testing of SYMBOL's memory; monitoring of SYMBOL; and decision making within DEBUG. These commands are summarized in Table III, where reserved mnemonics are shown in capital letters, and parameters and explanatory notes are given in lower case letters.

MAINTENANCE UNIT

To provide an independent maintenance and debugging capability, the System Tester was replaced by another AFU, the Maintenance Unit. Figure 5 schematically illustrates the final system configuration as delivered to Iowa State University. Input/Output equipment, controlled by a logic switch, is shared between the Maintenance Unit and a high-speed batch.
TABLE III—DEBUG Macros and Commands for Controlling the System Tester

Control of the general test environment

CLEAR TO ZERO, unordered list of registers.
COMBINE, page number, group number, word number, COMB. Address generator
EJECT. Skipped output listing to a new page
END. Signaled end of a testing session
EQUATE, old mnemonic = new mnemonic. Corrected simple keypunch errors
HELLO, user's identification.
LIST ALL. Permitted printing of everything
LIST ERRORS ONLY. Only error messages were printed
MOVE, destination = source. Register-to-register transfers
PAUSE, number of milliseconds. A timed delay.
PRINT, unordered list of registers.
REGISTER STATUS. Printed contents of interface registers
REPEAT LAST MEMORY COMMAND, number of times. With XADR = RADR each time
SET, register = value. Register initialization
SPLIT, source, SPLTP, SPLTG, SPLTW. Address decomposer
STOP. Press "START" on console to continue
WATCH, unordered list of hardware interface register mnemonics.

Control of the hardware interface

LOAD BUS AND STEP CLOCK, XDATA, XADR, OP. Single-step bus control
START CLOCK. Let SYMBOL free-run
STEP CLOCK AND PRINT CONTENTS OF MAIN BUS, number of times.
STOP CLOCK. Force SYMBOL to halt
SYSOFF. Contraction of "system off"
SYSON. Contraction of "system on"

Maintenance of files stored on control computer's disc

DELETE, FILE n. Deleted a previously saved file
FILE STATUS. Gave status of all files on disc
RESTORE, FILE n. Loaded SYMBOL memory and DEBUG status
SAVE, FILE n. Stored SYMBOL memory and DEBUG status

Loading, reading, writing, and testing of SYMBOL memory

FIND, beginning address, ending address, data, mask. Searched for data
INITIALIZE, first address, last address, datum. Loaded memory with datum
LIST, beginning address, ending address. Dumped memory on line printer
LOAD, beginning address, ending address, ordered list of data elements.
TEST CORE, XDATA, number of times. High-speed repetitive memory test
ZERO CORE. Cleared SYMBOL memory to all zeros

Monitoring of SYMBOL

PREPARE TO TRACE, unordered list of AFU names. Set up for trace mode.
TRACE, XDATA, XADR, OP, unordered list of conditions for exiting.

Decision making within DEBUG

GO TO, label. Skipped cards until label was found
IF, data, mask, condition, skip flag, number of cards. Decision maker
LABEL, user-chosen label. Provided target for a GO TO command

terminal. The Maintenance Unit console, with START, STOP, INHIBIT LIST, and RESET switches, is integrated into the SYMBOL console. A simple punched card input, easily produced at a keypunch or through another processor called SUPERBUG (written in FORTRAN for an IBM System /360) is used to direct the hardwired Maintenance Unit. To preserve the results of all prior testing, all DEBUG card decks were converted to equivalent SUPERBUG decks through a modified version of DEBUG which used the System Tester and SYMBOL in the translation process. The essential features of this control language are given
below in hopes that they will provoke additional ideas.

The Maintenance Unit is a card-driven character processor where each card contains one or more commands. The command format is space independent, but parameters are order dependent. The first non-blank character on a card is the command code. Parameters (if any) are separated from each other by a dash (minus sign). Two consecutive dashes (with zero or more intervening spaces) indicate an (optional) missing parameter, in which case the register contents remaining from the previous command are used. The first semicolon on a card terminates processing of further characters from the card (i.e., human-readable comments, sequence numbers, etc., may follow a semicolon) and the command is executed with parameters received before the semicolon (a semicolon is automatically generated at the end of a card if none were previously found). Because of minimized decoding logic, the only valid characters before a semicolon are the hex digits 0 through F, the dash, and the (ignored) space.

**Maintenance unit hardware**

Table IV describes the hardware registers contained in the Maintenance Unit. In the table and in the following descriptions, each upper case character in a register mnemonic represents a single card column.

During operation, YYYYYYYY and Q are cleared to zeros before each new command is read. RRRRRRR, RC, and DDDDDDDDDDDDDDDDD are preserved from one instruction to another unless changed by a memory command. M is preserved until a new value is loaded or until a system reset is performed, after which M = 0. XMP, CCCCCC, Ø, P, and TN are preserved from command to command unless specifically changed.

**Superbug commands**

Note that P, TN, Ø, Q, XMP, and CCCCCC are treated as a single ordered parameter when loaded from a card. P and TN may be changed without affecting Ø by following TN with a dash or semicolon (or by leaving the remainder of the card blank). Other than these registers, the Maintenance Unit contains no memory; card commands are processed "on the fly" without benefit of additional storage.

**Debugging Techniques**

Although the command and control languages for the System Tester and the Maintenance Unit differed somewhat (with the Maintenance Unit having a considerably smaller repertoire), debugging using the two systems was essentially the same. Both systems allowed the full range of Main Bus transfers to be initiated. This meant, for example, that the Memory Controller (the central hub for most information transfers) could be thoroughly and exhaustively checked without having any other AFU functioning. Control exchange cycles enabled the starting and stopping of any AFU or groups of AFU's. The environmental commands and the macros, such as ZERO CORE, provided a good human interface with the system and permitted easy setup of initial conditions in preparation for a test. And last, and definitely most important, was the ability of both testers to monitor or "trace" the operation of SYMBOL as it operated under its own steam.

In the System Tester, monitoring was accomplished by a PREPARE TO TRACE card (which set the interface into a trace mode) followed by a TRACE command with parameters to start the proper AFU(s) and to establish stopping conditions. A single card combining both setup and startup commands was used with the Maintenance Unit. In either case, once the monitoring process was begun, all (or any subset) of the events occurring on the Main Bus (plus several "patchable" lines used to assist in observing internal AFU operations) could be printed on the high-speed line printer. At the option of the user, printing could be suppressed and the results of the trace could be stored "round robin" in a small temporary buffer (on disc with the System Tester or in SYMBOL core with the Maintenance Unit) for later playback only if something crashed. The buffer in the System Tester held
TABLE V—Examples of SUPERBUG Commands which Control the Maintenance Unit

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3; Pause. Press “START” on console to continue processing.</td>
<td></td>
</tr>
<tr>
<td>4; Enable printing of results on the line printer</td>
<td></td>
</tr>
<tr>
<td>5; Inhibit printing for greater test throughput</td>
<td></td>
</tr>
<tr>
<td>7; Skip the line printer to the top of a new page</td>
<td></td>
</tr>
<tr>
<td>8; Load core with data</td>
<td></td>
</tr>
<tr>
<td>8; Load successive core locations with datum</td>
<td></td>
</tr>
<tr>
<td>9; Load beginning at address zero</td>
<td></td>
</tr>
<tr>
<td>9; Dump non-zero contents of memory on line printer</td>
<td></td>
</tr>
<tr>
<td>9; Dump non-zero memory beginning at address zero</td>
<td></td>
</tr>
<tr>
<td>A; Load core with data</td>
<td></td>
</tr>
<tr>
<td>A; Load successive core locations with datum</td>
<td></td>
</tr>
<tr>
<td>B; Test core with all ones and all zeros, in that order</td>
<td></td>
</tr>
<tr>
<td>C; Memory cycle without check of results</td>
<td></td>
</tr>
<tr>
<td>C; With check</td>
<td></td>
</tr>
<tr>
<td>D; System reset. Equivalent to pressing “RESET” on the console</td>
<td></td>
</tr>
<tr>
<td>E; System Supervisor transmit without trace</td>
<td></td>
</tr>
<tr>
<td>E; Disc-synchronized System Supervisor transmit</td>
<td></td>
</tr>
<tr>
<td>F; Set new system test mode</td>
<td></td>
</tr>
</tbody>
</table>

the last 1600 operations whereas the Maintenance Unit's buffer held but the last 64; usually the most recent 10 to 20 were sufficient to analyze whatever problem caused the crash. As might be expected, this "instant reverse playback" feature gained heavy use since testing without the real-time printing of results speeded things up manifold; only the last cycles which caused failure needed to be printed.

SUMMARY

The full-scale running SYMBOL system has demonstrated that:

1. A very high-level, general-purpose, procedural language and a multi-processing, multi-programming operating system can be implemented directly in hardware.
2. Design and construction techniques using only large two-layer printed circuit boards for all system interconnections, together with a functionally factored system result in an economical, serviceable, reliable, and "debuggable" system.
3. Effective testing techniques that connect an operating processor to an unproven system which has a bus-organized architecture can be developed to enhance and greatly simplify the debugging process.

ACKNOWLEDGMENTS

The author wishes to acknowledge the help of the past and present members of the Digital Systems Research Department at Fairchild Camera and Instrument Corporation who contributed to the testing phase of SYMBOL. Steve Lundstrom designed the original System Tester and contributed to the development of the DEBUG language. Tom Cook assisted with the implementation of the Maintenance Unit. Eric Palmer wrote the SUPERBUG processor. The guiding hands of Rex Rice and William R. Smith were present throughout the entire project. And, of course, the hard work and enthusiasm of the other members of the department made the project succeed.

REFERENCES

1. R. Rice W. R. Smith
   SYMBOL: A major departure from classic software-dominated von Neumann computing systems
   AFIPS Conf Proc 38 pp 575-587 1971
2. W. R. Smith R. Rice G. D. Chesley
   T. A. Lalioitis S. F. Lundstrom
   M. A. Calhoun L. D. Gerdoul T. G. Cook
   SYMBOL: A large experimental system exploiting major hardware replacement of software
   AFIPS Conf Proc 38 pp 601-616 1971
3. G. D. Chesley W. R. Smith
   The hardware-implemented high-level machine language for SYMBOL
   AFIPS Conf Proc 38 pp 563-573 1971
4. B. E. Cowart R. Rice S. F. Lundstrom
   The physical attributes and testing aspects of the SYMBOL system
   AFIPS Conf Proc 38 pp 589-600 1971
   Integrated complementary transistor nanosecond logic
   IEEE Proc 52:12 pp 1584-1590 1964
6. R. Rice
   Functional design and interconnections: Key to inexpensive systems
   Electronics 39:3 pp 109-116 1966