A flexible standard programming system for hybrid computation

by WOLFGANG GILOI, DIETER BECKERT and HANS C. LIEBIG
Technical University of Berlin
Berlin, Germany

Hardware structure of hybrid computer systems

The combined operation of analog and digital computers in hybrid computer systems requires a special hardware interface because of the different modes of operation and the different ways of data representation in both computers. Generally, in such a symbiosis all the functions of the analog computer, which are normally under manual control by the user, here have to be under control of the digital program, except for one case: once the digital program has started an analog computer run, the analog computer is on its own. In the case of combined simulation, i.e., if both computers operate simultaneously, from the viewpoint of the analog computer its digital partner now plays the role of a single, but very complex, computing unit to which some analog signals go and from which other signals come back.

From the viewpoint of the digital computer (and in the following we will always adopt this point of view), the analog computer and the interface are an entity representing an external process which has to be controlled. We assume that this process may include the various procedures listed in Table I, which all have to be managed by the digital program. (For notational convenience the word ‘program’ will always stand for the digital program, while we shall call the analog part of the entire program a ‘setup.’)

Table I
List of Procedures Which are Provided by the Hardware Interface

<table>
<thead>
<tr>
<th>Non-Time-Critical Functions of the Hardware Interface:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Control of the analog computer modes by the program</td>
</tr>
<tr>
<td>2. ‘Run Time’ selection (of one or more analog computer runs) by the program</td>
</tr>
<tr>
<td>3. Signalization of the actual start of an analog computer run (change of the integrator mode) to the program</td>
</tr>
<tr>
<td>4. Sensing of the current mode of the analog computer by the program</td>
</tr>
<tr>
<td>5. Signalization of the end of an analog computer run (‘run time elapsed’) to the program</td>
</tr>
<tr>
<td>6. Setting of potentiometers and subsequent checking by the program</td>
</tr>
<tr>
<td>7. Setting of analog switches by the program</td>
</tr>
<tr>
<td>8. Readout of analog outputs and potentiometer settings under control of the program and transfer of these data to the digital computer</td>
</tr>
<tr>
<td>9. Sensing of the state of Boolean variables occurring in the ‘logic box’ of the analog computer by the program</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time-Critical Functions of the Hardware Interface:</th>
</tr>
</thead>
<tbody>
<tr>
<td>10. Multiplexing and sampling of analog data and conversion into digital form; transfer of these data into the digital computer at an arbitrary rate (serially)</td>
</tr>
<tr>
<td>11. Transfer of Digital data into the digital-toanalog converters (in parallel), conversion (and, in certain circumstances, smoothing) of these data</td>
</tr>
<tr>
<td>12. Transfer of overload messages and other error messages to the program</td>
</tr>
<tr>
<td>13. Transfer of external interrupts (arbitrarily programmable on the analog computer) to the program</td>
</tr>
</tbody>
</table>

All the procedures of items one through nine have to be executed while the analog computer is in the HOLD, RESET, or STANDBY mode. Hence, they are not very time critical. On the contrary, the procedures 10 through 13 take place while both computers are running.
thus imposing on the program problems of real-time process management.

All this requires a digital computer that has process control computer capabilities such as a fast communication I/O-channel (preferably with direct memory access or a cycle-stealing mode, respectively), a hierarchic interrupt system, and control and sense lines. In addition to the real-time clock which is part of any analog computer, either the interface or the digital computer has to provide a second real-time clock (defining the ‘frame time’ yet to be explained).

In terms of hardware only, the interface (plus analog computer) appears to the digital computer like one more peripheral device, yet much more complicated than the customary peripheral equipment. A much higher complexity of the data flow to and from that device is mandatory, varying between single control bits or sense bits and words or blocks of words to be transferred at a very high rate. In terms of the system programming, however, this particular peripheral device represents a complicated process, occurring in real-time and manageable only if a specially designed software interface is provided, which is as powerful as the hardware interface.

The requirements for a powerful software interface and its structure

The machine languages of digital computers which have the above-mentioned process control computer capabilities include instructions that energize control lines or check sense lines, handle interrupts, activate the I/O-channel and prepare its interface hardware so that a block of data words can be put in or out, etc. However, it is much too tedious to program a hybrid computer that way (despite the fact that it has sometimes been done). In the case of the hybrid computer system about which this paper deals, a program written in machine code would require 35 instructions just for selecting an arbitrary analog computing element, and 25 more instructions to read the output of that element and transfer it into the digital computer. The setting of one potentiometer—including a subsequent check of the actual setting and the handling of possible error messages—takes about 150 instructions in machine code. Many of these assembly language instructions are not just simple, mnemonic words such as the arithmetical operation-code, but I/O-instructions which have to be specified, e.g., by a never-to-memorize six digit octal number. However, it is even worse if one leaves the subtle tasks of interrupt handling to the programmer.

All these problems may occur in process control applications too, but for such a special purpose, a program has to be written only once. Hybrid computer system programming, however, combines the short-term aspects of pure digital programming with the crucial difficulties of real-time data processing.

So, first of all, one has to eliminate coding by machine instructions. A comfortable software package for a hybrid computer system has to offer the possibility of writing any program in one of the customary problem-oriented languages that programmers are used to, for example, FORTRAN or ALGOL. Of course, such a language has to be augmented by a number of special subroutines. These subroutines take care of all the procedures listed in Table I. Furthermore, a special executive program is necessary which handles the synchronous operation of both computers and all the interrupts and error messages which may occur.

But that is not sufficient. Analog computer operators are used to having control of the entire system at any arbitrary time instant. They are able to interrupt the execution of a program in order to change potentiometer settings as well as the whole setup and restart or rerun thereafter the program. A hybrid computer system should provide the same possibilities, but this cannot be done while the system is under control of the digital program. Conversely, the program may sometimes need the help of the human operator (at least as long as the automatic patching problem has not been solved yet). The first demand to hand over the control of the system to a human operator or to receive it back at any arbitrary moment is unique. If this is granted, there are actually three parties involved alternating in the control of the system: the digital computer, the analog computer, and the human operator. This gives us additional reason for using the term ‘software interface’ as a supplement to the ‘hardware interface,’ meaning that only the proper design of both can make a hybrid computer system manageable. Additionally, some special utility programs are very helpful which perform an automatic checkout of the analog setup and of the entire system, thus detecting, indicating, and diagnosing errors and component failures.

Hence, a hybrid computer system ‘housekeeping’ software package should at least consist of the following programs (names have been assigned to the various programs which we can refer to):

1. HARTRAN (hybrid-procedure augmented real-time FORTRAN)
2. HYTROL (hybrid system control)
3. ACID (analog computer and interface diagnosis)
4. STATEST (static test of the analog setup)
5. HYBRID LIBRARY PROGRAMS

From the collection of the Computer History Museum (www.computerhistory.org)
HARTRAN denotes the set of all hybrid subroutines by which the problem-oriented language (in our case a special FORTRAN version) has to be augmented. If the software interface is organized the way it will be subsequently described, HARTRAN includes also the executive program. HYTROL provides the required man-machine-interaction. ACID is a necessary augmentation of the debugging programs which are already existing for the digital computer. STATEST enables the digital computer to check-out the analog setup and to detect and indicate set-up errors as well as component inaccuracies or failures. The HYBRID PROGRAM LIBRARY encompasses a number of hybrid standard programs which usually cannot be found in the standard program library supplied with the digital computer.

It is common practice to compose a complex programming system of several subsystems, because it gives a much better way to implement, improve, maintain, and change such a system. Doing so, one has to take care, of course, of all possible interconnections between the various subsystems. In order to facilitate the realization and the use of the subsystems, we define first of all a certain set of modules which are common components in all of them. The modules occur in three different forms:

PROGRAMMED OPERATORS (POPs)
STANDARD HARTRAN SUBROUTINES (SUBs)
SPECIAL HARTRAN FUNCTIONS (FUNCs).

A POP is a subroutine that has a mnemonic name, like any other macroinstruction, combined with a parameter by which the execution is defined. The SUBs are standard subroutines of the HARTRAN compiler, i.e., they don't have to be declared and can be called by a CALL statement together with their name and an arbitrary array of parameters. The FUNCs designate procedures by which the output of an analog component or a Boolean variable is fetched from the analog computer. The actual variable receives the name of that function, and under this identifier it can be a member of any arbitrary arithmetic or Boolean expression (like the standard functions in the usual FORTRAN language).

SUBs and FUNCs, for example, take care of all the procedures listed in Table I and others. POPs are first of all used within HYTROL and STATEST, and the ACID complex of test programs rely on SUBs as well as on FUNCs. The special library programs are written in HARTRAN. We will talk about these modules in more detail when we come to the description of the various subsystems.

What problem-oriented programming language should be used? The standard FORTRAN language (especially FORTRAN II) has some shortcomings. For example, it does not provide the possibilities of recursive call of subroutines, instructions for bit manipulation, and handling of interrupts. Notwithstanding, for two reasons, we selected FORTRAN. The first and most important reason was that FORTRAN is the most often used programming language. The intention of this paper is, however, not to describe just one of many possible implementations of a hybrid system software interface, but to propose a standard type of software interface which can be implemented on any system. Therefore, it has to be independent of particular assembler languages and it has to be based on the problem-oriented language most in use.

The second reason was an individual one. For our computer (SDS 930), a special FORTRAN version exists, called REAL-TIME FORTRAN, which does include all the above mentioned features, especially the possibility of connecting subroutines which respond to interrupts. Of course, this is no comfort to somebody who has to rely on the standard FORTRAN II version, but, even then, the construction principles outlined in this paper can be applied. HARTRAN would in this case stand for 'hybrid-procedure augmented FORTRAN,' and this programming system may lack some valuable conveniences, but it will still be feasible.

The organization of the software interface; HARTRAN and HYTROL

The two possible states of a hybrid system

From the discussion in the above section, the concept of organization of a hybrid computer software interface becomes quite obvious: The system as an entity (hardware and software) has two possible states, namely:

(I) the CONTROL state
(II) the RUN state.

In the CONTROL state a number of standard subroutines can be called within the framework of HARTRAN. The subroutines execute the special hybrid procedures which we have listed in Table I under items one through nine, and some more. Additionally, by calling a special subroutine by the name HYTROL, control is transferred to the console typewriter or any other arbitrary input device, enabling the human operator to interrogate or control the system. HYTROL, on the other hand, has to provide instructions by which a return to the HARTRAN program is possible.
Another most important HARTRAN subroutine called OPERUN switches the system from the CONTROL state into the RUN state. By the CALL OPERUN statement two different processes are started simultaneously:

a. An analog computer run is initiated (switching the analog computer from the STANDBY, INITIAL CONDITION or HOLD mode, to the OPERATE mode).

b. In the digital computer, a procedure called RUN is started that initiates a continuous scanning, sampling and converting of analog data, a manipulation of those data, and transfer of the digital results into the digital-to-analog converters.

At this point we have to comment about the two different classes of hybrid computation and their consequences on the organization of the data transfer between both computers. In the first class of operation, both computers operate alternatively, while in the second class, both computers execute a hybrid program simultaneously.

For operations of the first class, the programmer should be given the flexibility of a random input or output of analog data; i.e., each I/O procedure has to be programmed individually, and every time the multiplexer or DAC addresses can be arbitrarily chosen.

Unlike the first case, the second case may be extremely time-critical, especially if a combined simulation of a system with high natural frequencies has to be performed in real-time. Since in this class of operation an input or output sequence of analog data may occur hundreds or thousands of times during one computation run, it would be unnecessarily awkward and time consuming to program this by a random access which needs to declare for every input or output the multiplexer or DAC addresses. Therefore, a procedure should be available which performs automatically and periodically a certain I/O-sequence. The first address and the length of the input and output sequences have to be declared just once (in a program header). All the time during the execution of that program, a fixed sequence of multiplexer input lines are scanned and a fixed sequence of DACs are loaded periodically. If the channel has the capability to transfer data blocks autonomously to and from the memory in a cycle-stealing mode bypassing the CPU (we call this the interface feature), this particular I/O-mode provides not only a most efficient way of programming, but also the fastest possible way of execution.

HARTRAN subroutines are available for both ways of inputting and outputting analog data. It has to be emphasized that this way of programming can be used for any computer, while the actual subroutines, of course, depend on its specific structure. In the following, we shall find some more SUBs which are a result of the specific hardware structure of our hybrid computer system. But this is not contradictory to our claim of proposing in this paper a standard software interface. The only thing we have to try is to assume a hardware structure as general and flexible as could be. The SUBs and the corresponding POPs which may then be required for a less flexible system are a subset of what we shall define in the following. The particular subroutines which can be called as SUBs or POPs have to be written for the individual system anyway. Following this philosophy, for example, it does not matter whether or not the multiplexer inputs are equipped with parallel track-and-hold circuits, or whether the DACs have two buffer registers or only one. Of course, the corresponding subroutines have to be written differently, but the structure of the software interface and the programming language are not affected. The same holds for some unique hardware features of our system.

**The control state and HARTRAN**

In the following we shall list all the standard procedures which can be executed by a HARTRAN program. On the left side we list the respective format, and on the right side we give a short specification of the procedure. Furthermore, the SUBs are classified by their function and a comment is made on each class.

**Analog Computer (AC) Mode Control**

CALL CON :: the next run will be 'continuous operation' unlimited in time

CALL CON H :: the next run will be 'continuous operation' until the selected time has elapsed; after that the AC goes in 'hold' and continues when a new 'operate' instruction is received

CALL REP :: the next run will be 'repetitive operation'

CALL REPH :: the next run will be 'repetitive operation'; after the first run, the AC goes in 'hold' and starts the next repetition run only on receipt of a new 'operate' instruction

CALL ITR :: the next run will be 'iterative operation'
CALL ITRH :: see REPH and replace ‘repetitive’ by ‘iterative’

CALL STY :: if the AC was in ‘hold’ it is now switched to ‘reset’ or ‘standby’ (both are synonymous)

CALL SRTN (‘rt’, ‘ot’, ‘ht’) :: setting of the ‘normal’ run timer

CALL SRTC (‘rt’, ‘ot’, ‘ht’) :: setting of the ‘complementary’ run timer, ‘rt’ is reset time; ‘ot’ is operate time; ‘ht’ is hold time (all declared in milliseconds)

Comment: A model of the AC mode control is a (virtual) manual control that has six push-buttons and six thumbwheels. The push-buttons are labeled: continuous operation (CON), repetitive operation (REP), iterative operation (ITR), operate (OP), hold (H), and reset or standby (STY). By pushing one of the first three buttons the mode of the next computer run is prepared, and by pushing the OP button this run is actually started. If, in addition to REP or ITR, the button H is switched on, we have the ‘single run’ mode. The combination of CON and H interrupts the continuous operation after the selected run time has elapsed; the AC goes in the HOLD position from where it can be restarted in order to continue the operation. By the (virtual) thumbwheels, the (normal) ‘run timer’ can be set, defining the three phases of a repetition cycle individually. The ‘complementary run timer’ setting is only required for the iterative operation. Though this mode does not make much sense in a hybrid computer system, it is part of most analog computers and shall thus be taken into account.

Setting of Potentiometers and Function Generators

CALL POTSET (‘addr.u’, ‘value u’, ‘addr.v’, ‘value v’, . . .) :: setting of an (unlimited) number of potentiometers

CALL POTSETL (‘addr.u’, ‘value u’, ‘addr.v’, ‘value v’, . . .) :: setting of potentiometers with a printout of the actual addresses and settings

CALL FGSET (‘value 1’, ‘value 2’, . . .) :: setting of function generators; ‘value 1’, ‘value 2’, . . . are the (fixed) breakpoint values of the function given as a decimal fraction of the reference (e.g.: + .4875).

Comment: ‘addr.’ may be any combination of letters and figures, depending on the AC address system; ‘value’ is a 4-digit decimal number following the decimal point (e.g.: F 127, .0178). For the error messages see a later section (HYTROL). FGSET, of course, only makes sense if the AC has digitally settable function generators. (In our system, we developed these units ourselves.)

Control Lines

CALL SET (‘x’, ‘y’, ‘z’, . . .) :: The control lines listed as parameters of the expression are set to TRUE.

CALL CLR (‘x’, ‘y’, ‘z’, . . .) :: The control lines listed as parameters of the expression are set to FALSE (CLR = ‘clear’).

Comment: The control lines are used to set or reset flipflops in the AC logic box.

Selection of Analog Components and Senselines; Functions

POT (‘addr.’) :: The potentiometer with a named address is selected, and its output is a variable of the program identified by POT (‘addr.’)

AMP (‘addr.’) :: The amplifier (multiplier, function generator, etc.) with the named address is selected, and is a variable of the program identified by AMP (‘addr.’).

SL (‘x’, ‘y’, . . .) :: The logical values of the sense lines ‘x’, ‘y’, . . ., are composed by ‘and’ and the result becomes the logical value of the function.

Comment: As mentioned above, functions can be variables of arithmetical or Boolean expressions. One can write, for example, a FORTRAN IF statement as follows (x being another variable):

IF (AMP (47) − .5800 * X) 10, 10, 20

If ‘x’, ‘y’, ‘z’, are addresses of sense lines, one can write, for example, (without having to declare ‘x’, ‘y’, and ‘z’ as LOGICAL)

IF (SL(1) . OR . NOT. SL(2, 3) 1, 1, 2
where the expression of the IF statement is
\[ SL1 \lor SL2 \land \neg SL3! \]

**Entry to HYTROL**

CALL HYTROL :: The control and test program HYTROL is called and the system control is handed over to the human operator

**Frame Time Selection**

CALL FTS('time') :: In preparing the run state, the time of a digital computation frame (see next section) is selected by setting a special real-time clock ('time' = 3 place decimal number giving the time in milliseconds).

**DAC Mode Selection**

CALL MODC :: Simple conversion (the AC reference is also the reference for the digital-to-analog converters (DACs)).

CALL MODM :: The DAC reference voltage can be an arbitrary analog variable, hence, conversion is combined with an (analog) multiplication.

CALL MODI :: The DACs combine the conversion with a straight-line segment interpolation (or extrapolation).

*Comment:* It is another special hardware feature of our system that the digital-to-analog converters (DACs) can operate in three different modes (we designed them ourselves for that particular purpose). The first mode is just simple conversion. In the second mode the converters may be used as multiplying DACs providing the multiplication of the converted digital output with any arbitrary analog variable. A considerable number of analog multipliers may be saved, and the errors may be reduced (since the MDACs are more accurate than analog multipliers).

The output time function of a usual DAC is a 'staircase' function, i.e., the output voltage is constant as long as the digital input stored in the DAC register is not replaced by a new one. In the third mode, our interpolating (extrapolating) DACs replace the staircase function by a straight-line segment interpolation (or extrapolation) between the data points, resulting in a better (and smoother) approximation of a continuous signal of which the data points are samples. For an interpolation, the program has to calculate the increments between a current output and its successor. If the successor is not available to the program, the interpolation must be replaced by an extrapolation based on the current output and its predecessor. A variation of the time interval between two data points (= frame time) is automatically taken into account. If the digital-to-analog converters have only the DAC and MDAC modes, the MODI subroutine is immaterial; if there is only the normal DAC mode, all the SUBs of this section are immaterial. The mode specification holds for all DACs. If only part of them shall multiply, the others have to be patched to the reference. If only part of them shall interpolate, the mode is MODI, and the ones which have not to interpolate receive simply the increment zero. In our case, this is particularly simple, as we have a 24-bit word format where the first 16 bits are the value, and the last eight bits are the increment.

**Asynchronous Data Transfer**

CALL AD ('x', 'addr.x', 'y', 'addr.y', ...) :: transfer of data blocks from the AC to the DC. 'x', 'y', ... are variables or expressions of the HARTRAN program identified by the numbers of the multiplexer input lines by which the variables are fetched.

CALL DAC ('x', 'dest.x', 'y', 'dest.y', ...) :: transfer of data blocks from the DC to the AC. 'x', 'y', ... are variables or expressions of the HARTRAN program identified by the numbers of the multiplexer input lines by which the variables are fetched.

CALL DAI ('x', 'INCx', 'dest.x', ...) :: transfer of data blocks from the DC to the AC. 'INCx', 'INCy', ... are the increments of the variables 'x', 'y', ... between two data points; for the other parameters see CALL DAC.

*Comment:* DAC has to be used in connection with MODC or MODM; DAI is only used in connection with MODI.
Switch from the Control State to the Run State

CALL OPERUN :: The system is switched into the run state, either for starting an analog computer run only (with no simultaneous digital operations) or for a combined operation of both computers.

Comment: Most analog computers have the possibility to change all the integrator time constants simultaneously by a factor 10 (e.g., by a '10 times faster' or a '10 times slower' switch). If this should be done under control of the program, the OPE operator (respectively the OPERUN SUB) may include a parameter which specifies the respective integration rate. This has the advantage, that nothing else in the program has to be changed. We suggest this as a possibility but we don't feel that it makes much sense to have this parameter under control of the program.

The run state

Figure 1 illustrates the general scheme of the RUN procedure executed by the OPERUN subroutine. Right after the start of the procedure by a CALL OPERUN statement, the subroutine sends (after having executed some preliminary instructions) an OPE instruction to the AC, hence starting an analog computer run. When the AC integrators really start, the AC sends a "start" interrupt to the DC causing the beginning of the first computation frame. By these means we obtain well-defined timing of computers. Simultaneously, both real-time clocks (the 'run time' and the 'frame time' clock) are started.

Every time the selected frame time has elapsed, the 'frame time' clock sends an 'end of frame' interrupt (EFI) to the DC, indicating the end of a frame and the beginning of the next one. Each EFI starts a combined input/output of analog data blocks as illustrated in Figure 2. Length and formation of the respective block have been specified together with the various source and object addresses by calling the DADC or DADI subroutines to be explained in the following.

Eventually, the selected run time of the analog computer will have elapsed. The AC goes at that moment in the HOLD mode and sends an 'end of run' interrupt to the DC causing the program to switch back to the control state. The 'start' and the 'end of frame' interrupt are of the same priority (higher than the 'end of run' interrupt).

Some additional comments should be given on the organization of a computation frame. Each frame starts with an output procedure, whereas the outputs are the results of the computations in the preceding frame (in the case of the first frame they are initial values). This happens as fast as the digital computer can put out these data (in our system it takes two memory cycles = 3.5\mu s for each DAC). The DAC settling time (which in our case is about 0.5\mu s) is not relevant in this context, as in fact the DACs could be loaded simultaneously if the DC could do that.

Right after the output sequence follows the input sequence (for the current frame). Depending on the throughput rate of the multiplexer/converter the ADC may feed-in data at a lower rate than the DC could handle. In such a case, one can use the idle time between two inputs for performing the necessary fixed-point/floating-point transformation on the current input. While we started with this approach, we have now an ultra-fast multiplexer converter in our system that has a higher maximum throughput rate than the DC can handle. (This multiplexer/converter is at present the fastest commercially available device of its kind and was developed by ourselves. It has a 15-bit accuracy at a maximum throughput rate of 500,000 per second.) Therewith, we put in data at the maximum possible speed of the DC channel and perform the fixed-point/floating-point transformation later on. Though the total execution time is not reduced, the skewing errors become negligible, though parallel
track-and-hold circuits (which would, in turn, introduce some other inaccuracies) are not used.

After the output/input sequence and the associated fixed-point/floating-point transformations have been finished, the execution of the computation frame begins. Depending on the chosen frame time, a time interval follows during which the DC is idle until the EFI occurs. It is at least very difficult, if not impossible, to estimate the exact execution time a priori in order to choose the frame time so that any considerable idle time is avoided. If the computation frame, for example, includes branching instructions or external interrupt, the execution time of a frame may not even be constant. This dilemma is solved by a special procedure within the RUN subroutine which measures the idle time, (the DC is not actually idle, but counting clock impulses). The minimum idle time of a computer run is stored and may be printed on request. Thus, after one test run the programmer can determine the right frame time. If a negative idle time occurs (i.e., if the frame time chosen was too short) an error message is printed in any case.

The part of the program which follows the OPERUN statement and which constitutes what we call the 'computation frame' may include any regular FORTRAN expression as well as some particular SUBS. SUBs which can be called within this program are listed as follows:

**Synchronous Data Transfer to and from the AC**

CALL DADC
('DA', 'NDA',
'AD', 'NAD') :: data transfer in both directions synchronized by the EFI according to Figure 2.
DA : first memory address of the output data block
NDA : length of the output data block
AD : first memory address of the input data block
NAD : length of the input data block
The DACs have to be preset either to MODC or MODM.

CALL DADI
('DA', 'INC',
'AD', 'NAD') :: like CALL DADC but with the DACs in MODI.
'INC' : first memory address of the block of increments

*Comment:* CALL DADC and CALL DADI can only be called as part of the RUN subroutine. NAD = 0: only transfer D-to-A; NDA = 0: only transfer A-to-D. Error messages are given if illegal addresses or numbers of data words are specified.

**External Interrupts, Sense Lines and Control Lines**

CONNECT SUB 'no'
(FINT('no') . . .) :: SUB 'no' is the name of a subroutine written by the programmer which is sensitive to 'free' external interrupts (FINT), i.e., the subroutine is executed when the interrupt occurs. Notice that SUB 'no' is not a standard HARTRAN subroutine.

CALL SET ( ),
CALL CLEAR ( ), SL ( ) : See previous section

*Comment:* The sense line and control line SUBs are used as in the control state. To write particular subroutines which are sensitive to external interrupts and to connect them to the program by a simple CALL or CONNECT statement is only feasible in special FORTRAN versions.

**Return Into the Control State**

CALL HOLD :: The analog computer is switched from the OPERATE to the HOLD mode and the system goes into the control state.

CALL HYTROL :: HYTROL can also be entered from the run state, switching the system necessarily to the control state.

*Comment:* If an overload or a channel error (erroneous multiplexer or DAC addresses or ADC overload) occurs, an interrupt with highest priority is sent to the DC, causing the system to return into the control state. Either OVERLOAD or CHANNEL ERROR is printed, and after that HYTROL is automatically called giving the user an opportunity to check for the error reasons or to restart the program execution.

**Software error compensation**

Intrinsic error sources of a combined computer system are:

(i) the necessary sampling of analog values transferred to the DC and a subsequent imperfect
smoothing of the digital results which go to the AC (the DACs approximate the theoretically required ideal low-pass filter very poorly)

(ii) time delay between corresponding input and output vectors because of the execution time of the digital computation frame

(iii) time shift between the various components of input and output vectors because of the serial input/output.

We call the respective errors caused by these three sources the ‘sampling errors,’ the ‘delay errors’ and the ‘skewing errors.’ The skewing errors can only be avoided by appropriate hardware measures such as parallel track-and-hold circuits at the multiplexer inputs and double-register DACs, or—as in our case—by an I/O transfer rate that is so high that no appreciable errors occur. Some authors have suggested reducing sampling and delay errors by utilizing additional hardware in the analog computer, however, these methods are tedious and too expensive to be of practical use. Giloi has shown in an earlier paper that a much better error compensation can be obtained by a digital filtering algorithm which is part of the computation frame. This method reduces sampling and delay errors simultaneously by an arbitrary high degree and is very easy to implement.

It has been shown in the quoted paper that the error compensation is almost as good when using non-recursive filter functions as in the case of recursive functions, yet nonrecursive functions do not cause stability problems. Therefore, HARTRAN includes a ‘filter’ algorithm of the kind

\[ G(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + a_4 z^{-4} + a_5 z^{-5} + a_6 z^{-6} \]

which may be executed in the context of the RUN subroutine by the statement

CALL ECF

\( ('a_0', 'a_1', 'a_2', 'a_3', 'a_4', 'a_5') \) :: digital error compensation filter passed by all the output-bound data

The maximum order of the filter is six, but by setting some of the coefficients to zero any lower order may be obtained. One important parameter of the filter does not have to be declared, as it is the ‘sampling interval’ which is equal to the frame time and, hence, is known to the filter algorithm by the CALL FRA statement. If the frame time is changed, the filter subroutine takes this automatically into account. (For the values of the coefficients \(a_0, a_1, \ldots, a_6\) see Giloi’s paper.)

**The control state and HYTROL**

By calling HYTROL the system is put into the control state (no matter in which state it was) and the system control can be performed by the human operator in a conversational mode. The program begins the dialogue by typing HYBRID CONTROL and in the next line a C. That means that the user is asked to specify the control medium, i.e., the device by which he wants to communicate with the system. If he types TY1, he continues using the console typewriter. In the case of a spacious installation or, even more, if the digital and the analog part of the system are located in different rooms (as in our case), it is very convenient, if not inevitable, to have next to the analog computer console a second typewriter which is selected by typing TY2. Card readers (CR) and paper tape readers (PR) may also be used, but only in the case when the system returns eventually to the HARTRAN program (i.e., the last statement has to be either RET or BEG). If an L is added to CR or PR, the instructions are printed. Any other statement than one of the set \{TY1, TY2, CR, CRL, PR, PRL\} typed after the C causes a SYNTAX ERROR message as well as any other violation of the HYTROL syntax.

After that, the user can start any controllable function of the system. For this purpose he has a set of MACROS at hand.

For the analog computer mode selection we have the following set of MACROS

\{CON, REP, ITR, CONH, REPH, ITRH, STY\}

corresponding with the SUBs of an earlier section. An analog computer run in the selected mode is actually started by typing OPE. This is in correspondence to what the OPERUN subroutine is doing.

For the setting of a potentiometer, one has to type

\[ \text{POT 'name' 'value'} \]

(blanks are arbitrary). The reader may notice that the potentiometer address is now part of the name, as the POPs allow only to specify one parameter (the value, defined as in the POTSET SUB).

The same feature holds when the AC readout system is activated by a MACRO. Here we write

\[ \text{SEL 'name'} \]

whereas, again, ‘name’ is the analog component which
has to be selected (for example P 137, A 72). The output of the selected unit is converted and printed as a 1-place decimal number plus sign (e.g., +.dddd). Sense lines and control lines can be checked (CHK) or activated (SET/CLR), respectively, by the following set of MACROS

\[
\text{\{CHK \('\text{name}\'), SET \('\text{name}\'), CLR \('\text{name}\')\}}
\]

For the setting of the 'normal' and the 'complementary' run timers and the 'frame time,' we have the MACROS RTN'ddd',

\[
\text{OTN'ddd'}, \text{HTN'ddd'} \colon \text{ (reset time normal, operate time normal and hold time normal)}
\]

RTC'ddd' \(,\) OTC'ddd', HTC'ddd' \( : \) (reset time, operate time and hold time of the complementary phase)

FTS'ddd' \( : \) (frame time selection)

'ddd' stands in all cases for a decimal number which specifies the time in milliseconds.

Finally, we have two MACROS which provide a return into the HARTRAN program (which we may have left by a CALL HYTROL statement). When typing

```
BEG :: begin
```

the HARTRAN program is started anew at the beginning, while by typing

```
RET :: return
```

the HARTRAN program continues by executing the SUB which follows the CALL HYTROL statement (the point at which the program was left).

The execution of a procedure is actually started by the subsequent carriage return. If a comma is typed, the procedure represented by the MACRO in the preceding line is once more executed. HYTROL provides some error messages and some diagnoses. Possible error messages are

```
SYSTEM NOT READY :: if the analog computer and/or the linkage is not switched on
SYNTAX ERROR :: in case of any violation of the HYTROL syntax
OVERLOAD :: if the AC has only a common but no individual overload indication
```

Diagnosing error messages are

```
OVERLOAD 'name' :: if the AC has an individual overload indication, 'name' means the name of the overloaded component
ADDRESS ERROR :: this message follows a POT, SEL, CHK, SET, or CLR if the name in this statement is none of the legal names of all the analog components, sense lines, and control lines.
SELECTION IMPOSSIBLE :: in this case the name in the preceding MACRO was legal but the associated component is not existing in this particular installation.
SERVO SETTING IMPOSSIBLE :: follows a POT operator if the servo does not work
TRIED THREE TIMES, DIFFERENCE ± .ddd :: the servo works, but after three trials the difference between the nominal value and the actual setting is still more than 0.04 percent (this threshold is arbitrary). The maximum deviation is printed.
```

The reader may notice that by the above listed MACROS, any parameter of a HARTRAN program can be arbitrarily changed, and thus, all control and possibilities of on-line debugging are provided. If the user should decide to terminate the hybrid computation (because he found a mistake in the program or a component failure which cannot be repaired immediately, or for any other reason) he types.

```
.M
```

\( M \)
Flexible Standard Programming System

Hence, the monitor system of the digital computer is called. The DC may start to execute any arbitrary program (or a batch of programs) under control of the monitor. This program batch may have nothing to do with the hybrid program, or it may contain standard I/O programs which are used in order to process the results of the hybrid computation (e.g., printer, plotter, or display routines). If the HARTRAN program ends with the last two statements

```
CALL HYTROL
M
```

the system returns automatically into the monitor mode.

**Time-sharing of the digital computer**

With respect to digital computer utilization, hybrid computation is extremely inefficient and, thus, expensive. The setting of a servo potentiometer for example requires some seconds, the setting of some tens of potentiometers adds up to minutes, during which the DC is idle. If the user is in the process of debugging his hybrid program on-line, it is even worse. In the case of an undebugged digital program, on the occurrence of an error, this program is dumped out, and the programmer may think about his mistakes without locking the computer. The hybrid programmer, however, will start to use HYTROL (since he has this wonderful tool available) in order to find out what is wrong. During the many seconds or minutes of meditation, he will hardly release the system. The worst of all cases occurs when he must halt to change his analog setup (as analog computer users are very likely to do).

If the monitor is loaded together with a 'background' batch of programs, the user can at least immediately restart the execution of the background batch (by the M operator) when he realizes that for some reason he cannot continue with his hybrid computation. But there are two problems: Once he has lost the access to the digital computer it will become difficult for him to get it back. The second problem is that in a so-called balanced system, the DC has hardly core memory enough to accommodate the resident part of the monitor, at least one background program, possibly a compiler, and in addition the HARTRAN program and HYTROL. (OPERUN and HYTROL alone take about 2.7 K of core memory.)

If the installation includes a rapid access disc or drum, the second problem could be solved by swapping the hybrid program for the background process and vice versa. From there it is only a minor step to time-share the digital computer all the time the hybrid program is in the HYTROL mode. Since there are only two processes competing for CPU time, a relatively large time slice can be allotted to the background program without causing discomfort to the user of the hybrid system.

The simplest way to implement such a system is to modify HYTROL so that when HYTROL is called it sends the C message and calls the monitor right after that to start executing the background batch. After a certain time (e.g., one second) given by a real-time clock (e.g., that of the AC run-timer), the background process is interrupted and HYTROL asks whether the hybrid system operator has in the meantime completed a HYTROL statement. If the answer is yes, this statement is executed and the system returns to the background process, as it does directly if there was no HYTROL statement completed. If the statement is RET or BEG, the background process is swapped out and the entire hybrid program is swapped in from the disc. Therefore, the digital computer is only exclusively assigned to the hybrid process when the latter is in its real production phase. Notice that for the implementation of such a system only a small part of the entire program package—namely HYTROL—has to be modified, one of the advantages of the modular structure of our software interface.

**STATEST**

STATEST is a special program which executes automatically a static check of the analog setup, hence detecting patching errors as well as failures or poor performance of components. By virtue of a built-in optimization strategy, the best suited test values are automatically evaluated by STATEST. Everybody who is familiar with the static check procedure on an analog computer knows that this is a most crucial point. In the case of STATEST, however, all that has to be done is to give the program a description of the analog setup (the analog part of the hybrid program in treatment). As far as we know, these features are unique.

The setup description consists of a list of connection statements, one for each component, which are written in the form of pseudo-equations. On the left side of such an equation the name of the 'object' is listed the output of which has to be checked. On the right side, the names of all source elements are listed which contribute to the input of that particular unit, combined by the arithmetic expression which this unit is performing. The names of the various units correspond with the addresses of the analog readout system. Variables (represented by the name of their source) which are fed to inputs with gain factors different from one
have to be multiplied in the equation by the respective gain.

In the case of a multiplier, for example, which may have the name M19 and which may multiply the outputs of summer S46 and of function generator F3, we write

\[ M19 = S46 \times F3 \]

(spanks are arbitrary). If the function generator input comes from integrator 17, we write

\[ F3 = F3(17) \]

Hence, F3 denotes both the name of a component (on the left side of an expression) and the corresponding function (on the right side of an expression) which has to be specified by a table and which has the source element as argument. Potentiometers and digital-to-analog converters (DACs) are not source elements, but are listed as multiplicative coefficients of the associated variables, represented by their name and not by the value to which they are set. Thus the setting of potentiometers and DACs may be arbitrarily changed without any change in the connection statement list. The current settings can be found in the POTSET list which has to be read-in or typed-in anyway. Figure 3a gives an example for one particular branch of the setup. In this case, we have to write

\[ I17 = 10 \times I16' + P35 \times NR + 10 \times I17' + F3 + S23 \]

Notice that the notation may be recursive, if there is a feedback from the output of the object to its input, except in the case of integrators (as in our example). In the static check mode, all integrators are changed into summers, the output of which yields the (negative) sum of all inputs, while the proper integrator output is substituted by a test value taken from the reference directly or via a potentiometer, or even from other components such as, for example, DACs. This is indicated in Figure 3a, and in this case we have to extend the connection statement list by the following two statements

\[ I16' = P1 \times NR \quad \text{and} \quad I17' = P2 \times PR \]

etc. (PR = positive reference, NR = negative reference).

When STATEST is activated (after the complete setup description, including the POTSET and FUNCTIONS list, was put in), it switches the AC mode to ‘static check’ and starts setting the potentiometers to

\[ .05 \times \text{ref} \leq v_o \leq \text{ref} \quad (\text{ref} = \text{reference voltage}). \]

This may not be the case, and thus, STATEST has to take steps in order to enforce the validity of this condition. For this purpose, a search procedure is started which goes through all the branches of the tree which is a graphical representation of the input/output relations.
of the particular unit under consideration. (For the given example, such a tree is shown in Figure 3b. The nodes of the graph are denoted by the algebraic operations and the branches by the name of the variables.) The program checks all the potentiometers in the respective equation (the potentiometers that specify the test values included) to see which of them has maximum influence on the terminal node when being changed. Thereafter, the setting of the potentiometer for which the object output is most sensitive is changed until condition (*) holds.

As STATEST has to traverse through several branches in order to find such a potentiometer, the operations of the intermediate nodes indicate the direction of the change (e.g., in the case of a divider a coefficient of the nominator would have to be increased in order to increase the output of the unit while a coefficient of the denominator would have to be decreased, and vice versa).

In any branch of the tree, the search procedure ends at the first potentiometer or a 'primitive' like the reference. In any branch, up to three components can be taken into account (this is a matter of the available core memory). STATEST does not care if there are overloads in parts of the setup other than the particular one which is currently under consideration. Hence, the convergence of the procedure is always guaranteed.

Once condition (*) holds for the current object the outputs of all the source elements are measured too, and the nominal results of the algebraic expression represented by the connection statement is calculated and compared with the actual (measured) result. Any deviation exceeding a given boundary $e^*$ results in an error message together with a printout of the error magnitude. The parameter $e^*$ is individually evaluated by the program for each one of the respective objects as a multiple of a common error level $e$. Hence, the number and gain factors of inputs of the object are taken into consideration (e.g., a gain of 10 leads to an $e^*$ that is 10 times greater than $e$, etc.). The basic 'precision parameter' $e$ depends, of course, on the particular analog computer and has to be declared by the user.

Eventually, when the program has gone through all the connection statements, all potentiometers are reset according to the POTSET list. Since that procedure is double-checked by the POTSET SUB, it is made sure that no new errors are introduced after the STATEST procedure has been finished. There are some more important details of this very sophisticated program which cannot all be mentioned in this paper. Our current STATEST version takes 10K of core memory for the interpreter plus some additional memory space for the connection statement list (approximately 1K for a typical 100-amplifier-program). But it could easily be shortened to run on an 5K configuration. STATEST includes also a routine which searches for hidden algebraic loops and gives a message if there is one. Furthermore, it can be easily connected with a program that calculates the setup (such as APACHE®). Of course, STATEST is only loaded into core when needed.

**ACID and the PROGRAM LIBRARY**

Most of the interface and analog computer functions could be checked by using HYTROL, but not all of them. Furthermore, if one wants to check out systematically the entire system and its reliability, it would be much too tedious to do it that way. All these tasks have to be automatically accomplished by special test programs. Naturally, the number and kind of such programs depend on the specific structure of the analog computer and the hardware interface, so that it is not possible to suggest a general concept as in the case of the operating system. Notwithstanding, we will give in the following a list of the ACID programs available and a short description of their function in order to indicate what these programs are about. It is almost needless to say that these programs use the same modules (SUBs, POPs, and FUNCs) as all the other parts of the software package.

**SELEC** :: Test of the AC selection system (ACSS). Selects for a designated number of times every AC component. Each time the actual contents of the ACSS address register is checked via a sense line feedback. In case of a deviation, an error message is printed together with the actual address and number of iteration. It is also printed if the selection of a component is impossible e.g., if this component does not exist.

**MODE** :: Test of the AC mode control. All possible AC modes are activated, and the actual mode is checked by sense lines.

**CLOCK** :: Test of the AC run timer and the frame time clock. Both timers are set so that the ratio of the run time with respect to the frame time is an integer greater than one. The number of end-of-frame interrupts (EFIs) which fall between a 'start' and a 'end of run' interrupt is counted. If this number does not correspond with the time ratio, an error diagnosis is given. The procedure is executed for various run time and frame time clock settings.
READ :: Test of the AC readout system. Test values are given via a DAC into the AC readout system and read back via the ADC into the DC (the AC readout system has a special dummy address for this purpose). Deviations which exceed a designated level are printed together with the DAC and multiplexer address used.

POT :: Repeated test of potentiometer settings. Arbitrary parameters are the number of repetitions, the increments of the pot settings and an error threshold. The program sets every pot to a certain sequence of values starting with zero and increasing by the chosen increment (until .9999). This procedure is repeated for a selected number of times. Eventually, a list is printed listing the name of the pot and the number of the iteration for each in which the error threshold has been exceeded. Additionally, the relative frequency, mean and variance of all listed errors are printed.

RUNI :: Test of the system interrupts under different AC modes. The number of EFIs is counted which occur in a given time interval and the content of the interrupt cells is checked. Errors are diagnosed.

LOGIC :: Test of sense lines and control lines (via closed loops which have to be patched on the AC's logic box)

IOAR :: Test of the I/O-address register in the interface and the multiplexer and converter address decoding.

LOOP :: Test of the data transfer to and from the AC. The DAC outputs are connected with the multiplexer inputs. Various values are given out via the DACs and received back via the multiplexer—ADC. Any difference between the original value and its received echo which exceeds a preselected error threshold is printed with the statistics of all the trials as in POT.

MULT :: Test of pairs of MDACs. One MDAC is set in increments of .01 between 0 and 1. Its output is multiplied by a second MDAC (which has a constant digital input). Nominal and actual results are compared. Errors are listed, together with all required information.

IDAC :: Test of the interpolating DACs.

It is even less possible to give a general concept of the required LIBRARY PROGRAMS as these depend essentially on the requirements of the users. Programs for parameter optimization routines, function storage and reproduction (with and without delay), fast FOURIER transform, statistical parameter estimation, etc., etc., should be mentioned.

It should also be mentioned that writing the digital part of a simulation program is in our case facilitated by a block oriented digital simulation language which we have developed and which we call SIESTA. SIESTA is a superset of the SCI-CSSL Language, and our SIESTA compiler was one of the first implementations of CSSL.

ACKNOWLEDGMENT

The authors would like to express their grateful appreciation to the Messrs W. Franke (who wrote OPERUN and most of the SUBs), W. Woletz (who wrote HYTROL), and R. Siol (who wrote most of the ACID programs) for their very valuable contribution to the implementation of the system. They are also indebted to Mr. M. E. Connelly, M.I. T. Electronic Systems Laboratory, for his helpful and encouraging suggestions.

REFERENCES

1 W GILOI Hybride Rechnersysteme Telefunken-Zeitung Jg 39H 1 1966 82-100
3 W GILOI Error-corrected operation of hybrid computer Proc 5th International Conference of AIAC Lausanne 1967
4 SCIENTIFIC DATA SYSTEMS Real-time FORTRAN II manual
5 W WOLETZ HYTROL—ein Verkehrspogramm für hybride Rechenanlagen Diploma Thesis Techn University of Berlin Inst of Inf Proc 1968
6 W FRANKE Ein Betriebssystem für hybride Rechenanlagen Diploma Thesis Techn University of Berlin Inst of Inf Proc 1969
7 G BEKEY W KARPLUS Hybrid Computation Chapter 7 J Wiley and Sons Inc New York 1968
8 P RECHENBERG
SIESTA—A CSSL—implementation
Internal Report Techn University of Berlin
Inst of Inf Proc 1968

9 The SCI continuous system simulation language (CSSL)

SIMULATION Vol 9 No 6 December 1967
10 C GREEN H D'HOOP A DEBROUX
APACHE— Breakthrough in Analog Computing
IRE Trans EC-11 October 1962 699-706