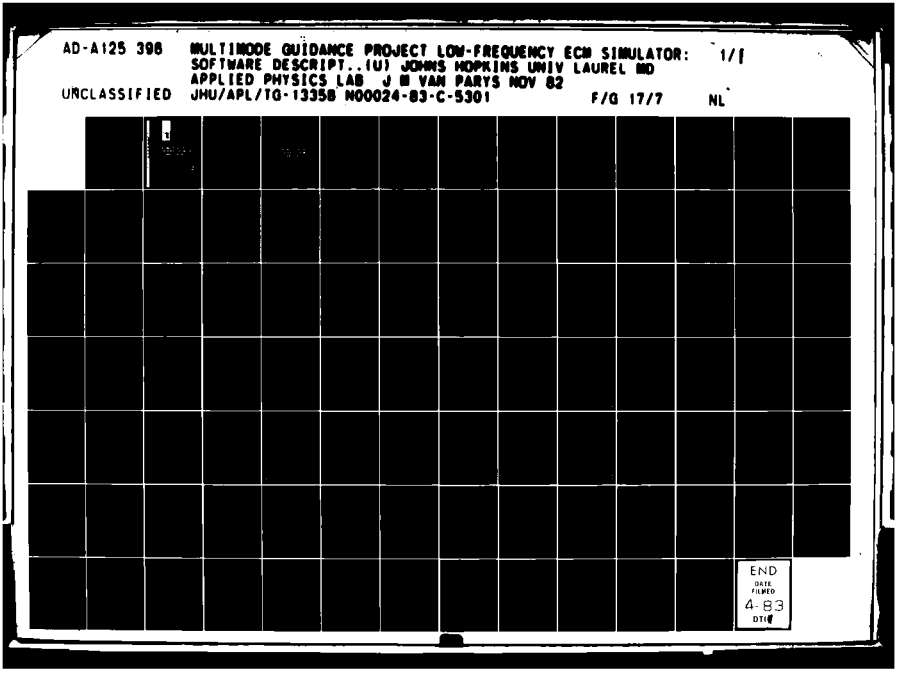
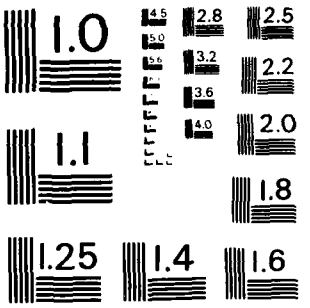
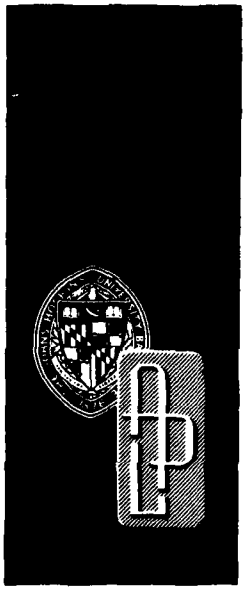
AD-A125 398 MULTIMODE GUIDANCE PROJECT LOW-FREQUENCY ECM SIMULATOR: 1/1
SOFTWARE DESCRIPT..(U) JOHNS HOPKINS UNIV LAUREL MD
APPLIED PHYSICS LAB J M VAN PARYS NOV 82
UNCLASSIFIED JHU/APL/TG-1335B N00024-83-C-5301 F/G 17/7 NL



END
DATE FILMED
4-83
DTIC



JHU/APL
TG 1335B
NOVEMBER 1982
Copy No. 9



(12)

Technical Memorandum

MULTIMODE GUIDANCE PROJECT LOW-FREQUENCY ECM SIMULATOR: SOFTWARE DESCRIPTION

J. M. VAN PARYS

THE JOHNS HOPKINS UNIVERSITY ■ APPLIED PHYSICS LABORATORY

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

PLEASE FOLD BACK IF NOT NEEDED FOR BIBLIOGRAPHIC PURPOSES

REPORT DOCUMENTATION PAGE

1 REPORT NUMBER	2 GOVT ACCESSION NO	3 RECIPIENT'S CATALOG NUMBER
JHU/APL TG 1335B	AD-A125 390	

4 TITLE (*and Subtitle*)

MULTIMODE GUIDANCE PROJECT LOW-FREQUENCY ECM SIMULATOR: SOFTWARE DESCRIPTION

5 TYPE OF REPORT & PERIOD COVERED

Technical Memorandum

6 PERFORMING ORG. REPORT NUMBER

TG 1335B

7 AUTHOR (s)

J. M. Van Parys

8 CONTRACT OR GRANT NUMBER (s)

N00024-83-C-5301

9 PERFORMING ORGANIZATION NAME & ADDRESS

The Johns Hopkins University Applied Physics Laboratory
Johns Hopkins Road
Laurel, MD 20707

10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

A3G0

11 CONTROLLING OFFICE NAME & ADDRESS

Naval Sea Systems Command
SEA-62R55
Washington, DC 20362

12 REPORT DATE

November 1982

13 NUMBER OF PAGES

93

14 MONITORING AGENCY NAME & ADDRESS

Naval Plant Representative Office
Johns Hopkins Road
Laurel, MD 20707

15 SECURITY CLASS. (*of this report*)

Unclassified

15a DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited.

17 DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18 SUPPLEMENTARY NOTES

19 KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

ECM simulator
Multimode guidance
Computer-controlled simulator
Multiple jammer environment
UHF jamming simulator
L-band jamming simulator
S-band jamming simulator

20 ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

The Multimode Guidance (MMG) Project, part of the Army/Navy Area Defense SAM Technology Prototyping Program, was established to conduct a feasibility demonstration of multimode guidance concepts. Prototype guidance units for advanced, long-range missiles are being built and tested under MMG Project sponsorship. The Johns Hopkins University Applied Physics Laboratory has been designated as Government Agent for countermeasures for the project. In support of this effort, a family of computer-controlled ECM simulators is being developed for validation of the contractor's multimode guidance prototype designs. The design of the low-frequency ECM simulator is documented in two volumes. This report, Volume B, describes the software design; Volume A describes the hardware design. The computer-controlled simulator can emulate up to six surveillance frequency jammers in B through F bands and will be used to evaluate the performance of home-on-jam guidance modes in multiple jammer environments.

DD FORM 1 JAN 73 **1473**

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

JHU/APL
TG 1335B
NOVEMBER 1982

MULTIMODE GUIDANCE PROJECT LOW-FREQUENCY ECM SIMULATOR: SOFTWARE DESCRIPTION

J. M. VAN PARYS

THE JOHNS HOPKINS UNIVERSITY ■ APPLIED PHYSICS LABORATORY
Johns Hopkins Road, Laurel, Maryland 20707
Operating under Contract N00024-83-C-5301 with the Department of the Navy

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

ABSTRACT

The Multimode Guidance (MMG) Project, part of the Army/Navy Area Defense SAM Technology Prototyping Program, was established to conduct a feasibility demonstration of multimode guidance concepts. Prototype guidance units for advanced, long-range missiles are being built and tested under MMG Project sponsorship. The Johns Hopkins University Applied Physics Laboratory has been designated as Government Agent for countermeasures for the project. In support of this effort, a family of computer-controlled ECM simulators is being developed for validation of the contractor's multimode guidance prototype designs. The design of the low-frequency ECM simulator is documented in two volumes. This report, Volume B, describes the software design; Volume A describes the hardware design. The computer-controlled simulator can emulate up to six surveillance frequency jammers in B through F bands and will be used to evaluate the performance of home-on-jam guidance modes in multiple jammer environments.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL, MARYLAND

CONTENTS

ILLUSTRATIONS

TABLES

1. INTRODUCTION

The Multimode Guidance (MMG) Project, part of the Army/Navy Area Defense SAM Technology Prototyping Program, was established to conduct a feasibility demonstration of multimode guidance concepts. Prototype guidance units for advanced, long-range missiles are being built and tested under MMG Project sponsorship.

The Johns Hopkins University Applied Physics Laboratory has been designated as Government Agent for countermeasures for this project. In support of this effort, a family of computer-controlled ECM simulators is being developed for validation of the contractor's multimode guidance prototype designs.

The design of the low-frequency ECM (electronic countermeasures) simulator is documented in two volumes. Volume A describes the hardware design of the simulator; Volume B describes the software design. The computer-controlled simulator can emulate up to six surveillance frequency jammers in B through F bands and will be used to evaluate the performance of home-on-jam guidance modes in multiple jammer environments.

Computer control allows the simulator to flexibly combine modulations into new patterns and to accurately repeat test frequencies. In particular, complex time-varying techniques may be created and repeated when needed. In order to take full advantage of the desktop computer that comprises the simulator controller, proper software is necessary. The simulator software may be considered in three parts: the user's program, subroutine blocks, and data.

The user's program determines the output that results when a test program is run on the controller. A user's program can be written for each new test form, and each such program can give many different outputs. User programs may be constructed to allow a wide range of operator inputs.

The user's program determines the simulator output through some well-defined sequence of steps, i.e., setting center frequencies, noise spots, pulse periods, etc. While each test program would have a unique collection and order of such steps, the steps them-selves will be common to all programs. The user's program carries out the steps through a series of program calls to a common group of subroutines. These subroutines are called subroutine blocks to distinguish them from any subroutines written by the user as part of a particular program.

Subroutine blocks handle common tasks, freeing the user from having to rewrite the necessary instructions for such tasks every time a new test program is written. The subroutine blocks can be used as building blocks in creating a new test program. Data formatting and addressing are handled by the subroutine blocks, using values stored as data.

Data used by the subroutine blocks allow the user to specify the output in user-friendly terms such as frequency and bandwidth. The subroutine blocks will then use the stored data to convert a user-specified parameter to the form needed by the hardware. Other data may be defined as needed by the user.

2. SIMULATOR DESIGN PARAMETERS

The reader must have some understanding of the MMG ECM simulator hardware in order to follow the description of the simulator software. This section will briefly summarize the simulator hardware design and parameters. For a more detailed discussion, the reader should consult the hardware documentation (Ref. 1). Appendix A of this report contains a block diagram of the simulator (Fig. A-1).

The simulator essentially consists of three major parts: the controller, the multiprogrammer, and the RF channels. In addition, there are arbitrary waveform generators, auxiliary modulation switch matrices, level set attenuator drivers, distribution boxes, and power supplies. All of the present low-frequency system (LFS) components except the controller are mounted in a rack frame.

The controller is an HP9825S desktop computer, with a number of ROM (read-only memory) options and a total of 22,910 bytes of useable memory (the controller may be upgraded to the 9825T standard or even changed to a different model such as the HP9826 without affecting the software's basic design logic). The controller communicates with the other simulator devices over the IEEE-488 bus. There are direct bus connections between controller and multiprogrammer and between controller and arbitrary waveform generators; the controller is indirectly connected to other parts of the simulator through the multiprogrammer.

The multiprogrammer consists of an HP6942 multiprogrammer and an HP6943 extender. Any general reference in this report to the HP6942 as the multiprogrammer may be understood to include the HP6943 as a subservient part. The multiprogrammer contains a number of slots holding digital cards of various types, such as digital output cards, digital to analog cards, etc. The controller sends data through the multiprogrammer to an addressed card in some particular slot, causing those data to be passed to whatever is connected to the card. The path could be reversed, with the data on a card being sent back to the controller. This would particularly apply to use of the digital input card. In the present LFS version of the simulator, the data flow is essentially one way, from the controller to the output devices, and no input measurements are made by the controller.

Of particular interest are the tune cards and the RF channel function control cards. Each tune card holds two 8-bit words that specify the status of an RF channel's tune frequency D/A converter and hence specify the tune frequency. The tune D/A number must be sent to the correct half of the appropriate card without affecting the other half of the card. Each channel function control card holds a 16-bit word that determines the status of devices within an RF channel, as follows:

Bit 15	: selected VCO,
Bits 12-14	: biphase circuit control,
Bits 9-11	: pulse circuit control,
Bits 6-8	: fill oscillator attenuation,
Bits 3-5	: noise generator attenuation, and
Bits 0-2	: noise video bandwidth.

Other multiprogrammer cards control the level set attenuators and the auxiliary modulation switch matrices, handle D/A conversion for auxiliary AM or FM, provide a pulse signal source from the timer/pacer card, and handle memory data (in the present LFS version, the memory cards are not used). For example, the digital output card in slot 11 determines if the FM auxiliary modulation matrix is off (latched) or on, and if on, what RF channel and what source are connected during the on state. Table 1 lists the multiprogrammer cards and their purposes.

The six RF channels generate the actual frequency outputs. Each channel has the same functional layout. There are two VCOs in each channel, one of which will be active (connected to the rest of the channel circuitry) at any time. The tune center is determined by the voltage from the tune D/A. This voltage into the VCO may be modulated by the fill oscillator, noise generator, and auxiliary FM. The VCO output may be successively modulated by linear AM, biphase, and pulse circuits. Level set attenuators set the output power amplitude reference level. There is a coupler to allow the RF channel output to be monitored from the front panel of the rack.

There are two types of RF channels in the present LFS version of the simulator, with a third planned. Type I covers the B and C frequency bands. Type II covers the D, E, and F bands. Table 2 summarizes the major simulator parameters. Type III, when

[1] H. M. Kaye, *Multimode Guidance Project Low Frequency ECM Simulator: Hardware Description*, JHU/APL TG 1335A (Oct 1982).

Table 1 – Multiprogrammer card slots.

Slot No.	*Card Type*	*Card Purpose*
0	Digital memory	1st part of memory pair
1	Digital memory	2nd part of memory pair
2	Digital output	Tune, RF channels 1, 2
3	Digital output	Tune, RF channels 1, 2
4	Digital output	Tune, RF channels 1, 2
5	Digital output	Function control, RF #1
6	Digital output	Function control, RF #2
7	Digital output	Function control, RF #3
8	Digital output	Function control, RF #4
9	Digital output	Function control, RF #5
10	Digital output	Function control, RF #6
11	Digital output	Level set control
12	Digital output	AM & FM auxiliary switch matrices
13	Timer/pacer	Pulse source (50% square wave)
14	Counter	Input pulse counting
15	Extender	Connect 6942, 6943
100	D/A	AM, RF #1
101	D/A	AM, RF #2
102	D/A	AM, RF #3
103	D/A	AM, RF #4
104	D/A	AM, RF #5
105	D/A	AM, RF #6
106	D/A	FM
107	Digital input	Input measurements

Note: Slots 0-15 are in the HP6942 multiprogramer; slots 100-107 are in the HP6943 extender.

available, will cover the G, H, and I bands and a small part of the J band.

There are two W175 arbitrary wave form generators available as part of the simulator. These devices may be used to generate almost any voltage waveform that can be defined as a function of time. One may be used for FM or pulse, the other for AM or pulse. The waveform generators are directly set by the controller; their outputs must be switched in through the multiprogrammer to affect the RF channel outputs.

The simulator output devices may be set independently, and outputs are formed by combining the effects of several devices. Unique test patterns can be generated by having the controller change device settings according to some desired scheme. Any test program would essentially be concerned with determining the hardware status of the simulator at any time during that test.

Table 2 – Major simulator parameters.

Parameter	*Specification*
Frequency coverage	
Type I, VCO A	250-500 MHz
VCO B	500 MHz; 1 GHz
Type II, VCO A	1-2 GHz
VCO B	2-4 GHz
Frequency accuracy	8 bit (256 steps) over VCO range (accuracy within 8 MHz; calibration dependent)
RF power	
Type I	+20 dBm, maximum
Type II	+17 dBm, maximum
Dynamic range	81 dB (1 dB steps)
Modulation types	
FM	
Noise	Gaussian or non-Gaussian, video bandwidths of 1, 10, 100 kHz; 1, 5 MHz
Fill	100 kHz square wave
Auxiliary	Sources: arbitrary waveform generator, D/A card, or external.
AM	≥55 dB maximum range, DC to 50 kHz rates Sources: arbitrary waveform generator, D/A card, or external
Biphase	5, 10, or 20 MHz comb; 10, 20, or 40 MHz psuedorandom noise
Pulse	PRF 500 kHz, maximum Sources: 10, 100 Hz (50% square wave), timer/pacer card (50% square wave), either W175, external

3. SOFTWARE FUNCTIONAL DESCRIPTION

Basic Organization

The controller is the key to the simulator's ability to output complex test waveforms. For time-dependent modulations, the controller can accurately control the sequence and timing of program steps.

Throughout this report, the user is understood to be anyone who uses the information in the report to write a test program. The operator is anyone who runs a test program. The user and operator need not be the same person.

A user's test program is essentially concerned with controlling the ECM simulator status during the test. In doing so, the test program specifies the test format (i.e., the simulator status at each step). Each test format requires its own unique test program, which must be provided by the user.

The user's program may directly specify the parameter values used in a test or it may allow an operator to select them. In the former case, the controller can prompt an operator to enter the test's parameter values and then verify that they are legal. The controller can also provide the operator with a list or menu of parameters, the operator entering values for only those that differ from a set of standard values. When an illegal entry is made, the controller can inform the operator and reprompt for a new value. Also, the user's program can record the parameter values for test documentation.

The controller's software has been organized so that the user may specify the simulator's status using output-oriented parameters such as frequency or bandwidth, instead of specific hardware parameters such as D/A number or attenuator setting. A test program will use a number of subroutines common to all test programs. These subroutines are called subroutine blocks in order to distinguish them from other subroutines prepared by a user for a specific test program. The subroutine blocks will handle the details of a particular step in a test format, freeing the user to concentrate on the test format itself. Subroutine blocks will accept output oriented parameters and find and set the corresponding specific hardware parameters. Data and device settings will be properly addressed using the subroutine blocks, without requiring the user to have a rigorous understanding of the controller's internal setup.

The subroutine blocks are used to build up a test format by setting devices and running sequences through the controller. The test format is determined by the user who is responsible for writing a program that organizes and uses the subroutine blocks as needed. If required, the user's program may set output devices without using the subroutine blocks but should not normally need to do so.

Subroutine blocks are independent in that the data set by one subroutine will not affect data set by others, unless several subroutines affect the same device or when one subroutine block calls upon others. For example, the pulse circuit switch of a given RF channel can be directly set without affecting any other settings, while the pulse circuit switch will automatically be reset properly when a subroutine block is used to set up either of the arbitrary waveform generators or the timer/pacer card as a pulse source for that RF channel. Independent subroutine blocks make it easier for a user to prepare complex test modulations in a simple building-block fashion.

The subroutine blocks require conversion data in order to get the hardware settings that correspond to the output-oriented parameters specified by the user. Such data are held in well-defined tables; examples are VCO tuning curves, noise spot attenuation tables, arbitrary waveform generator voltage curves, etc. Other data may be defined by the user for a specific test; such data would typically consist of parameter values to be passed on to the subroutine blocks.

The basic organization of any test program will be a hierarchical allocation of the available controller memory among the user's program, the subroutine blocks, and the data. The user's program uses the subroutine blocks to specify certain tasks, and the subroutine blocks use the data in carrying out those tasks. The organization of the user's program depends on the test involved, as indicated below.

Basic Programming Steps

Test Definition

As the first step in any test program, the user must be able to define the purpose of the test. The purpose of the test should in turn suggest the general sort of modulation pattern to use. Test definition is closely related to form definition (below), and, in practice,

the two steps will often be carried out simultaneously. The distinction is that test definition treats the entire simulator system as a black box out of which the test designer wants some particular modulation pattern, while form definition is used to set up the user's program in the controller. Test definition may be carried out by a test designer who then directs the user to program the controller for that test; the test designer need not be directly concerned with the actual program.

Form Definition

With a specific test definition as the goal, the user must form a program to carry out the test. This will involve determining at every part of a test the output device changes to be set and the parameter values of those changes. The user's program handles tasks in three areas: initialization, parameter determination, and output setting. Initialization would be a fairly standard procedure followed after power is turned on and a test started. Most of the user's work load in preparing a test involves parameter determination, with output setting being handled through the subroutine blocks.

In order to determine parameter values, one must know what parameters are needed, which amounts to knowing what devices should be set. The user should find it helpful to consider the test as a sequence of intervals, where each interval change is marked by some notable change in the simulator status.

During each interval, the user's program needs to explicitly handle only those devices whose status changes from their status in the previous interval. Other devices would remain as they were previously set. A number of changes closely timed can be considered as one interval. How close such timing must be is up to the user; the point of an interval structure is to make it easier to translate a test design to actual programming.

In setting up intervals, the user should be aware of the distinction between a set type and a run type output. A set type subroutine (which should be a subroutine block) will use the controller to set some other simulator device, with the resulting state remaining in force until the device involved is explicitly changed. A run type subroutine (which may be a subroutine block, or a modified one, or a new one written by the user) will use the controller to run a modulation pattern. The pattern may involve rapid, timed device changes that would tie up the controller. To illustrate this, consider an example of a hopping noise spot. The noise spot, power level, carrier signals, and the like can be handled by set type subroutines. The hop can be handled by having the controller rapidly change the center frequency of the VCO used through a run type subroutine.

The distinction between set and run types is important in setting up a sequence of intervals. A run type output ties up the controller so that no other tasks can be carried out while such an operation is running. When the controller stops running such an operation and moves on to its next task, the run type output will end. The beginning and end of run type outputs provide suitable interval divisions. No run type output from one interval will continue into a following interval. When several set type outputs and a run type are used in an interval, the set type subroutines are obviously called first.

Timing between intervals (and within the steps of a run type output) is provided by the controller. Timing control would typically use the wait instruction, making allowances for the time it takes the controller to carry out its operations. Timing control may also involve the multiprogrammer's real-time clock. The user may allow an operator to control interval timing through the controller's keyboard.

Appendix B contains more information on the basic programming steps.

Parameter Determination

Parameter values may be found in a number of ways. The ways can be distinguished according to whether literal numbers or variables are used in the actual subroutine calls and preliminary calculations; in the case of the latter, there are three ways of getting the value of the variable contents. Any combination of ways may be used in a program. The ways are:

1. Direct number
2. Variable
   a. Program
   b. Data file
   c. Operator entry

A direct number is easy to use but tedious to change. Variables (including expressions) make it easier to run the same test form repeatedly with different values on each run. Variables may be assigned direct numbers within a program. This is similar to use of direct numbers since it requires program modi-

fications in order to change the values, but by collecting all assignments in one part of the program it becomes easier to make such changes than if numbers were scattered throughout the program. Variables may also be assigned through the use of some predefined and taped data array prepared by the user. Such arrays allow a test's parameter values to be easily changed by changing the file contents loaded from tape.

More particularly, the user may set up a program to allow an operator to specify parameter values at the time of the test run. Operator inputs can be checked by the program. Several subroutine blocks also make it easier to handle operator inputs. Default values should be provided in case the operator does not wish to change a typical value; this will reduce the operator's workload and save testing time. The controller can prompt the operator with type and range information for each input.

Appendix C contains more information on parameter determination and passing.

User Subroutines

User Defined

The user will find it advantageous to make heavy use of subroutines when preparing a new, complicated program. Use of subroutines written by the user can make it easier to follow the structure of a program and to modify that program for later use in a different test form. Also, by using subroutines written according to a few general rules (chiefly involving internal variables used within the subroutine), the user can build up a library of new subroutines to join with the subroutine blocks as a source of prewritten program building blocks. This should be particularly useful when the user prepares any new run type output subroutines.

Label names can be used to positively identify a subroutine, including its purpose. The only restrictions on the labels available to the user are that the label must fit on one program line and must be unique (see Table 3 for a list of labels already in use by the subroutine blocks). Labels may also be used elsewhere in a program to set off the structure and to handle program branching.

The actual purpose and form of user-defined subroutines are up to the user and will depend on the test definition. Appendix D contains more information on user-defined subroutines.

Table 3 – Subroutine block labels.

?	initial
fval#	stepmod
fset	*stepval
fnoise	*stepwt
fnout	ownswp
pulse	swp175
biph	AM175
auxmod	DC175
AMaux	T/P
ampset	special
AMown	*valspec
*AMval	err stp
setVCO	shutoff
enter	inRFid
–	loadY$
	loadX$
	loadW$

* User-provided nextval subroutine (see Appendix D).

Predefined

The user may be required to write subroutines to carry out a predefined task in order to use certain subroutine blocks (see Table 4). The subroutine blocks in the lefthand column of Table 4 are run type subroutines that find a data value by calling on other subroutines, format the data, and send it to the proper address. The subroutines in the righthand column of Table 4 provide the next value to be sent out in sequence by the subroutines that call them. For convenience, the subroutines that return the data are called nextval subroutines.

Nextval subroutines must be provided by the user and must return the necessary data through a specified variable. The actual output of a calling subroutine is largely determined by the nextval. By changing the nextval, the user can easily change the output, without having to rewrite all the necessary addressing or formatting instructions handled by the calling subroutines. A nextval may take any number of forms, from complicated calculations to simple lookup of a table prepared in advance.

Appendix D contains more information on nextval subroutines.

Data

The subroutine blocks primarily use local dynamically allocated p-numbers as internal variables, with

Table 4 – Nextval subroutines.

Subroutine Block	*Nextval Subroutine/Returns**
AMown	AMval/dB of attenuation
stepmod	stepval/D/A code number stepwt/dwell at value (ms)
special	valspec/double tune word single tune word 3 channel function control words

* See Appendix D for return limits.

a few global variables (those visible to the user's program) for loop indices and nextval returns. Most global variables used by the subroutine blocks hold data tables used when converting output-oriented parameters to hardware-oriented ones (see Table 5).

The data tables must be dimensioned and loaded at the start of a test. The noise, fill, and W175 (FM) tables may need to be updated from tape storage when the active VCO in an RF channel changes or when the noise video bandwidth changes. Subroutine blocks ("load-$") will handle such updating.

The subroutine blocks use the following global variables and flags:

simple : U through Z
array : X, Z
string : U through Z
flag : 14

As a rule of thumb, the user should not assign new values to any of the variables just listed. They may be read at any time. All other global variables (and flags 0-12) are available to the user. Flag 14 is set and unset by most of the subroutine blocks; if the user wants flag 14 set in the user's program, the flag must be reset after calling a subroutine block that affects the flag or the blocks must be modified.

Appendix E contains more information on data and variables.

Error Handling

The subroutine blocks will carry out a number of simple checks on the parameter values passed to them. The checks are chiefly to ensure that a particular subroutine block's parameters are legal and within range. Checks in one subroutine block are independent of checks in other blocks. If an error is found, the subroutine blocks have no general provision for prompting an operator to correct an error, and use of defaults could lead the user and operator into misinterpreting the status of the simulator. When an error is found, the subroutine blocks will therefore set a coded number in the variable Z to indicate the cause of the error and then branch to a routine that reports the error, shuts off the simulator output, and stops the program. This requires a con-

Table 5 – Required dimensioned variables.

Label	*Purpose*	*Where/How Loaded* **
Z$[12,54]	Tune frequency data	File 2
X$[6,120]	Noise generator data	File 6, "loadX$"
Y$[6,120]	Fill oscillator data	File 5, "loadY$"
W$[6,120]	W175-FM data	File 4, "loadW$"
V$[120]	Tape-controller data transfers*	– –
U$[36]	Calibration identification, W175 outputs, operator input subroutines*	File 91
X[14]	Long-term constants	File 3
Z[22]	Card words	– –

* May be used by user's program if care is taken to avoid overwrite conflicts.

**Tape files are all on track 1.

scious effort by an operator to restart the test, which helps ensure that the operator will be aware of the error that must be fixed.

The user can avoid such controlled crashes by checking parameter values before calling the subroutine. It would then be possible for the user to define how to recover from a faulty parameter. Notably, the user's program can check operator entries, and if an error is found the program can reprompt the operator.

Appendix F contains more information on error handling.

Ancillary Devices

Tape

The HP9825 controller has a magnetic tape cartridge that will store programs and data on two tracks. About 20% of the track 1 is reserved in a definite format to hold data for the subroutine blocks (see Table 6); track 0 and the rest of track 1 are available to the user. It is recommended that file 0 of track 0 on program tapes be used to store an index and guide to the rest of the tape contents.

Appendix G contains more information on the use of the tape.

Table 6 – Tape track 1 reservations.

Files 1 to 91 on track 1 are reserved for the subroutine block data, as follows:

File No.	*Use*
1	Transfer program
2	Z$ (tune data)
3	X[*](constants table)
4	W$ (initial load, W175 data)
5	Y$ (initial load, fill oscillator data)
6	X$ (initial load, noise generator data)
7-18	Fill oscillator data
19-90	Noise generator data
91	Calibration identification

W175 Arbitrary Waveform Generator

The simulator has two W175 programmable arbitrary waveform generators. They may be programmed for any voltage waveform that can be defined as a function of time. A number of standard waveforms are available in ROM; arbitrary ones may be stored in PROM (programmable read-only memory) or RAM (random access memory). Output rates for a full W175 waveform block may range up to 19.5 kHz (higher if a partial block is used); the output modulation rate may be different from the W175 block rate, depending on the voltage amplitude.

It is the responsibility of the user to avoid allocation conflicts. One W175 may be used for FM or pulse and the other for AM or pulse; the user must avoid using either W175 for both purposes simultaneously. The waveforms and voltages for the two applications are not compatible. Normally this is not a problem and can be handled implicitly in the way a program runs. Explicit status tracking may be necessary if an operator controls the sequence of test intervals.

Appendix H contains more information on the use of the W175 waveform generators.

User Documentation

The user should document any new test program that is significantly different from existing programs, is likely to be used by others, or is considered a major test program. Programs can be partly self-documenting through good use of labels. File 0 on track 0 of a program tape can contain a brief guide to the test program. The controller's internal printer may be used to document test parameter values.

Written documentation should identify any new subroutines that may be useful in other programs, identify data files and requirements, and identify the tape files that hold parts of the program. Also, if operator inputs are accepted, the user's documentation should include an operator's guide detailing the available options at each input. A program listing should always be taken and given for hard-copy reference.

Appendix I contains more information on user documentation.

Miscellanous Information

There are a number of small corrections and modifications that could be made in the subroutine

blocks. These are detailed in Appendix K. None would significantly change the subroutine block logic.

Appendix L contains a program checklist.

Appendix M contains short miscellaneous information that the reader may find useful.

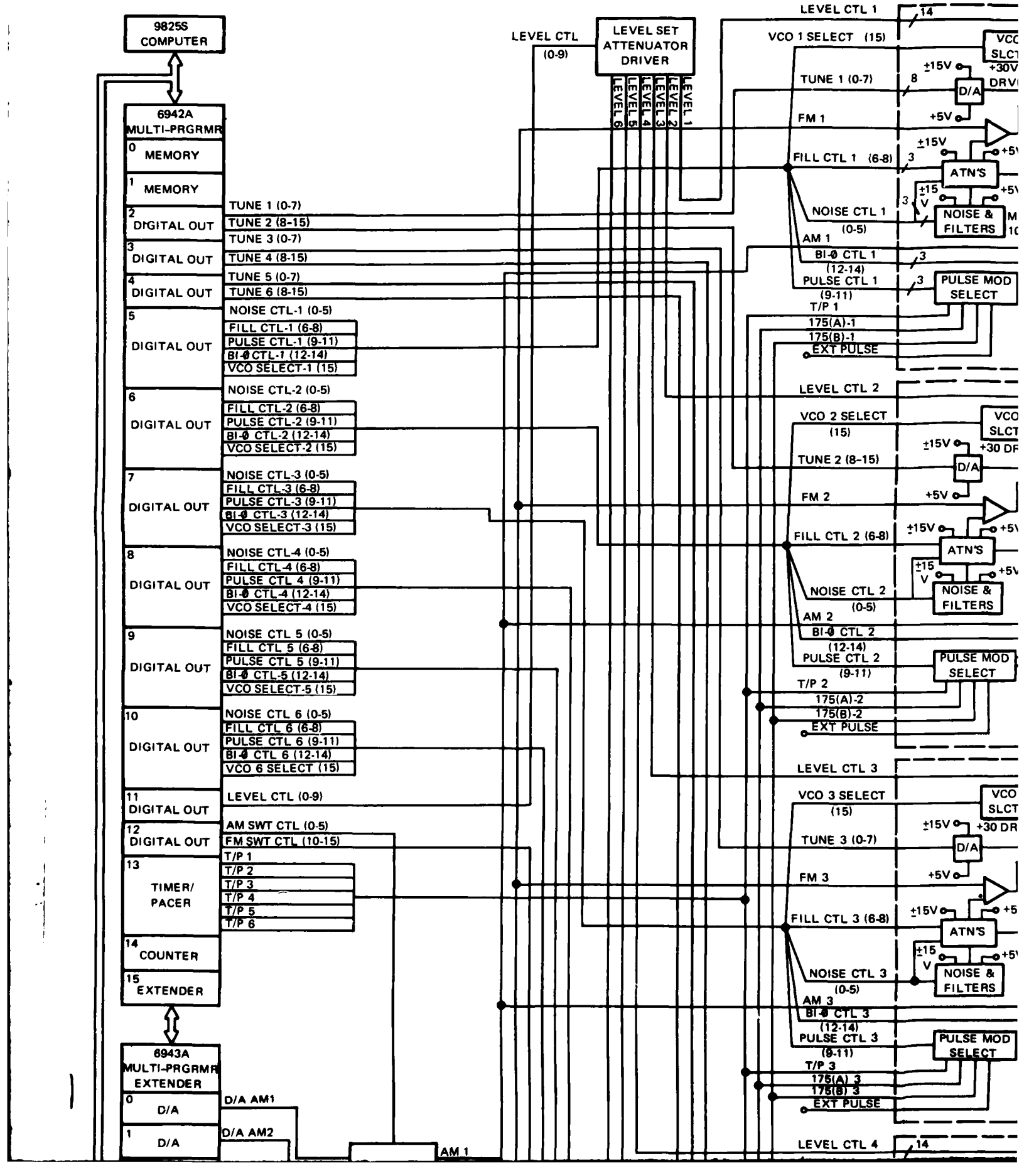
Appendix N contains one-line descriptions of the subroutine blocks, with more detailed descriptions in Appendixes O and P.

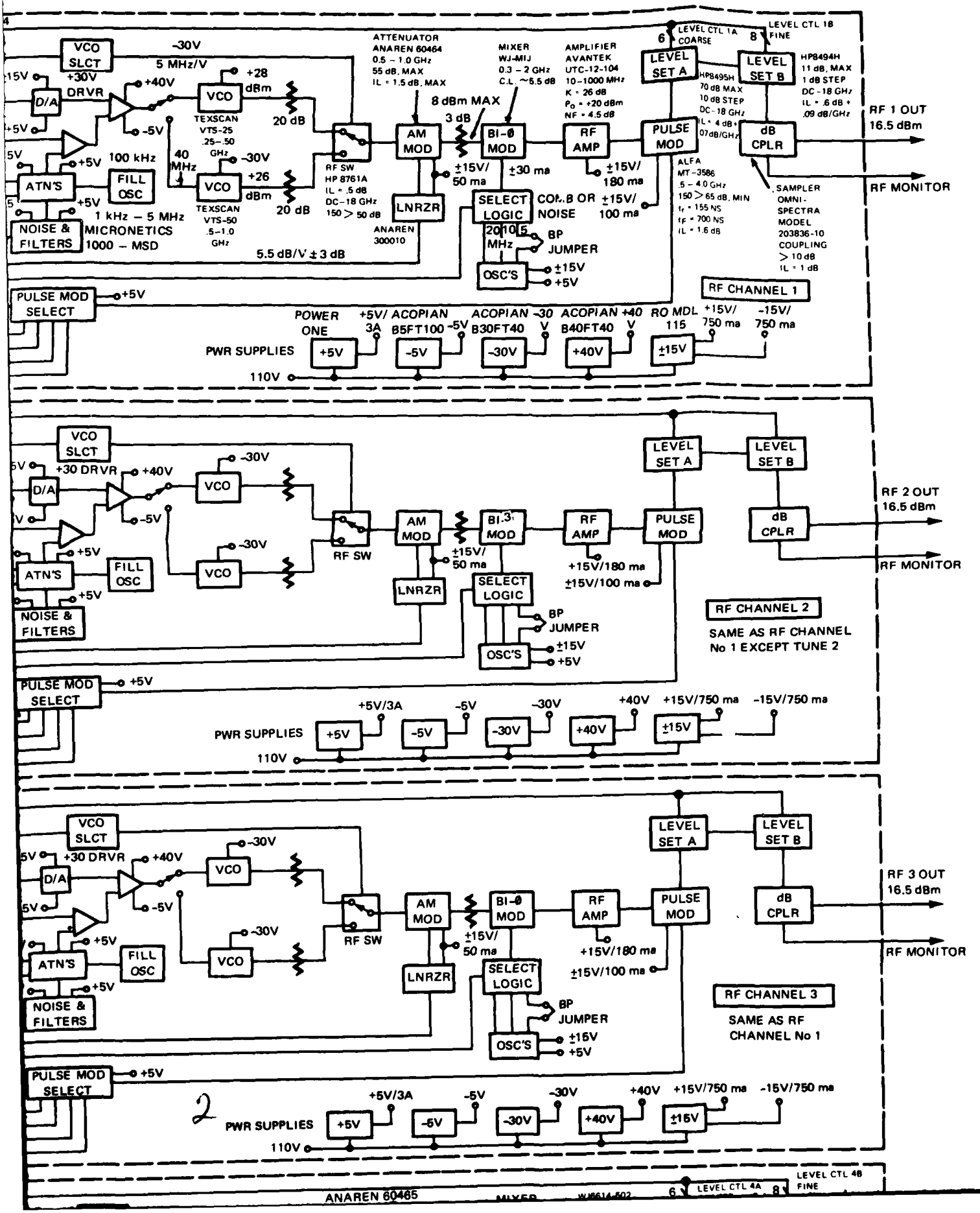
A number of programs have been prepared using the subroutine blocks; these will be described separately. In particular, a demonstration program has been prepared that will allow an operator to arbitrarily set up simple set type outputs in any order on any RF channels. Other programs will handle data calibration and hopping noise spots. It is planned to develop a library of test programs and program elements (such as subroutines) so that the user would find an existing program to handle a test.

ACKNOWLEDGMENTS

The author would like to acknowledge the help and advice of J. H. Braun, H. M. Kaye, and M. M. Soukup, who have given valuable suggestions and comments. The author would also like to especially thank M. E. Gibson, who typed the many drafts of this report.

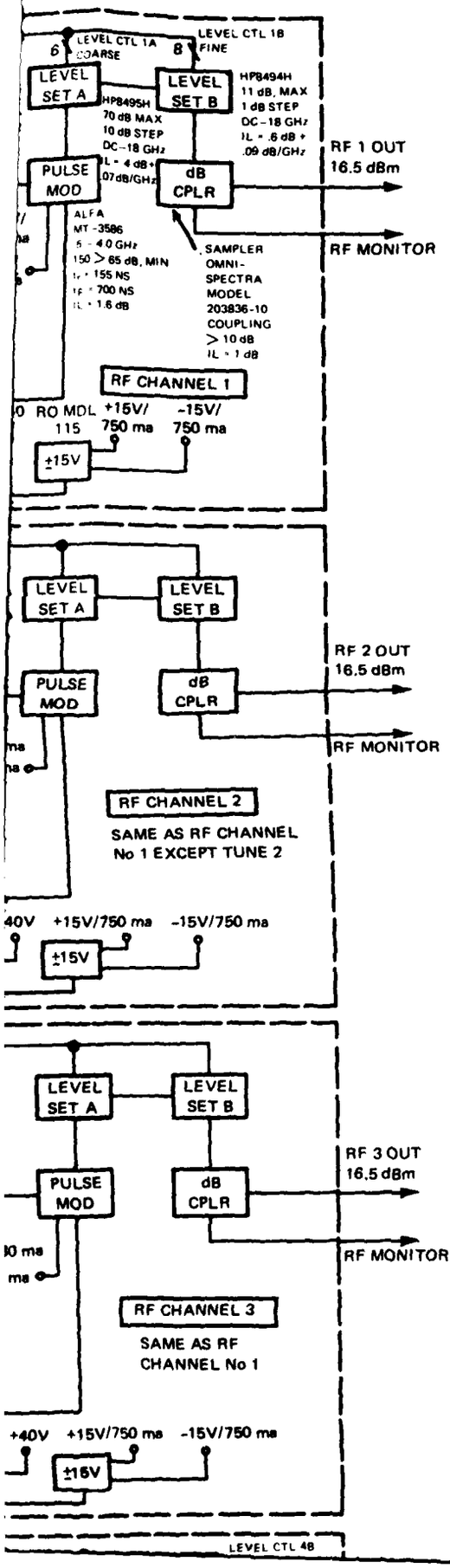
THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL, MARYLAND

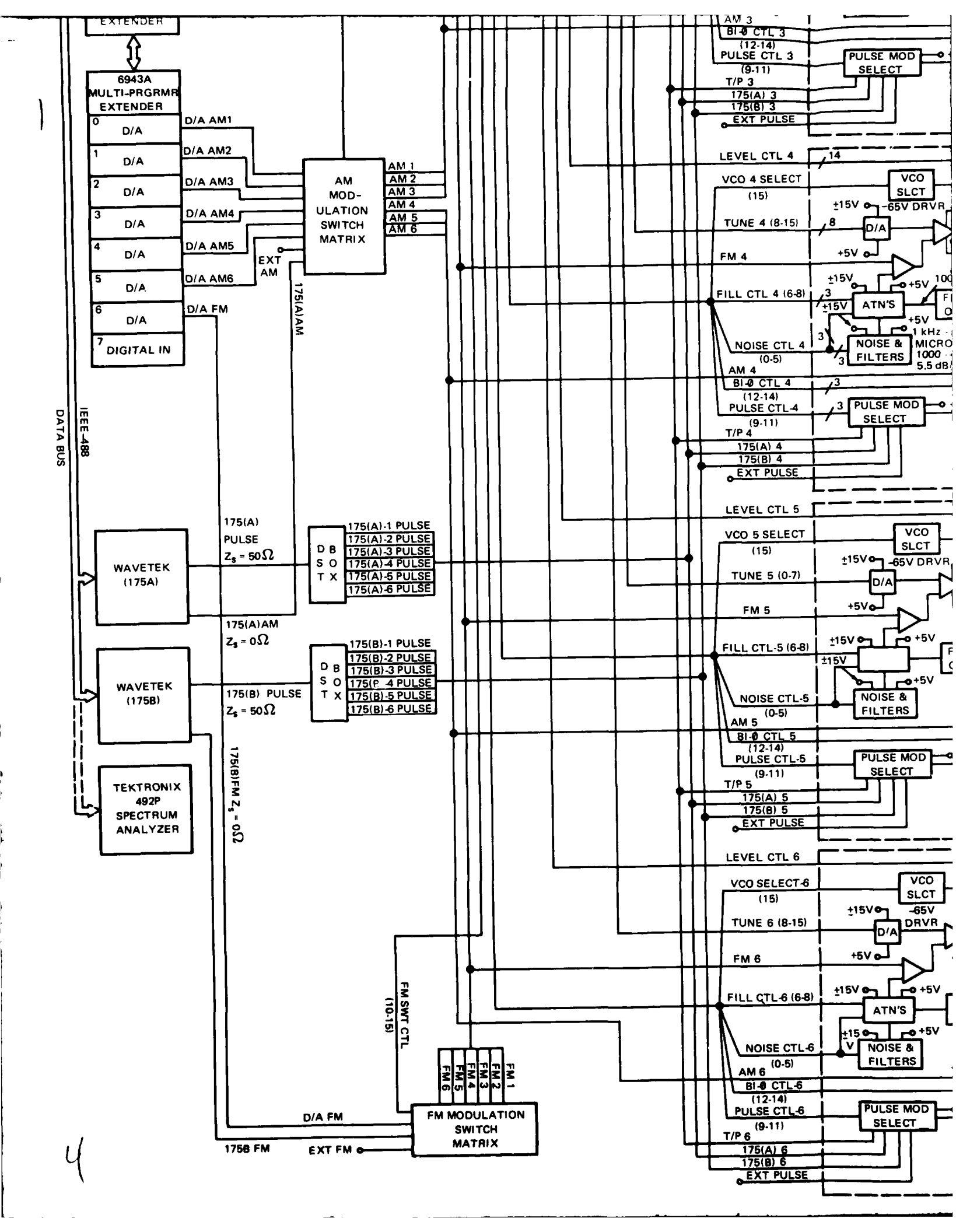



VCO SLCT
-30V
5 MHz/V
+30V
DRVR
D/A
+40V
-5V
VCO
+28 dBm
20 dB
TEXSCAN VTS-25 .25-.50 GHz
40 MHz
-30V
+26 dBm
20 dB
TEXSCAN VTS-50 .5-1.0 GHz
100 kHz
FILL OSC
ATN'S
NOISE & FILTERS
1 kHz - 5 MHz MICRONETICS 1000 - MSD
5.5 dB/V ± 3 dB
RF SW HP 8761A IL = .5 dB DC-18 GHz 150 > 50 dB
ANAREN 300010
ATTENUATOR ANAREN 60464 0.5 - 1.0 GHz 55 dB, MAX IL = 1.5 dB, MAX
AM MOD
8 dBm MAX
3 dB
±15V/ 50 ma
LNRZR
MIXER WJ-MIJ 0.3 - 2 GHz C.L. ~5.5 dB
BI-Ø MOD
±30 ma
SELECT LOGIC
20 10 5 MHz
BP JUMPER
OSC'S
±15V
+5V
AMPLIFIER AVANTEK UTC-12-104 10-1000 MHz K = 26 dB Po = +20 dBm NF = 4.5 dB
RF AMP
±15V/ 180 ma
COMB OR NOISE
±15V/ 100 ma
LEVEL CTL 1A COARSE
LEVEL SET A
LEVEL CTL 1B FINE
LEVEL SET B
HP8495H 70 dB MAX 10 dB STEP DC-18 GHz IL = 4 dB + .07dB/GHz
HP8494H 11 dB, MAX 1 dB STEP DC-18 GHz IL = .6 dB + .09 dB/GHz
PULSE MOD
ALFA MT-3586 .5 - 4.0 GHz 150 > 65 dB, MIN tr = 155 NS tf = 700 NS IL = 1.6 dB
dB CPLR
SAMPLER OMNI-SPECTRA MODEL 203836-10 COUPLING > 10 dB IL = 1 dB
RF 1 OUT 16.5 dBm
RF MONITOR
PULSE MOD SELECT
+5V
RF CHANNEL 1
POWER ONE
+5V/ 3A
ACOPIAN B5FT100
-5V
ACOPIAN B30FT40
-30 V
ACOPIAN B40FT40
+40 V
RO MDL 115
+15V/ 750 ma
-15V/ 750 ma
PWR SUPPLIES
110V
RF SW
RF 2 OUT 16.5 dBm
RF CHANNEL 2
SAME AS RF CHANNEL No 1 EXCEPT TUNE 2
RF 3 OUT 16.5 dBm
RF CHANNEL 3
SAME AS RF CHANNEL No 1
ANAREN 60465
LEVEL CTL 4A
LEVEL CTL 4B FINE

APPENDIX A

BLOCK DIAGRAM OF SIMULATOR




EXTENDER
6943A MULTI-PRGRMR EXTENDER
0 D/A
1 D/A
2 D/A
3 D/A
4 D/A
5 D/A
6 D/A
7 DIGITAL IN
D/A AM1
D/A AM2
D/A AM3
D/A AM4
D/A AM5
D/A AM6
D/A FM
EXT AM
AM MOD-ULATION SWITCH MATRIX
AM 1
AM 2
AM 3
AM 4
AM 5
AM 6
175(A)AM
DATA BUS
IEEE-488
WAVETEK (175A)
WAVETEK (175B)
TEKTRONIX 492P SPECTRUM ANALYZER
175(A) PULSE $Z_s = 50\Omega$
175(A)AM $Z_s = 0\Omega$
175(B) PULSE $Z_s = 50\Omega$
175(B)FM $Z_s = 0\Omega$
DBSOTX
175(A)-1 PULSE
175(A)-2 PULSE
175(A)-3 PULSE
175(A)-4 PULSE
175(A)-5 PULSE
175(A)-6 PULSE
175(B)-1 PULSE
175(B)-2 PULSE
175(B)-3 PULSE
175(B)-4 PULSE
175(B)-5 PULSE
175(B)-6 PULSE
FM SWT CTL (10-15)
FM 1
FM 2
FM 3
FM 4
FM 5
FM 6
FM MODULATION SWITCH MATRIX
D/A FM
EXT FM
175B FM
AM 3
BI-Ø CTL 3 (12-14)
PULSE CTL 3 (9-11)
PULSE MOD SELECT
T/P 3
175(A) 3
175(B) 3
EXT PULSE
LEVEL CTL 4
VCO 4 SELECT (15)
VCO SLCT
±15V
-65V DRVR
TUNE 4 (8-15)
D/A
+5V
FM 4
FILL CTL 4 (6-8)
ATN'S
NOISE CTL 4 (0-5)
NOISE & FILTERS
1 kHz
MICRO
1000
5.5 dB
AM 4
BI-Ø CTL 4 (12-14)
PULSE CTL-4 (9-11)
T/P 4
175(A) 4
175(B) 4
EXT PULSE
LEVEL CTL 5
VCO 5 SELECT (15)
TUNE 5 (0-7)
FM 5
FILL CTL-5 (6-8)
NOISE CTL-5 (0-5)
AM 5
BI-Ø CTL 5 (12-14)
PULSE CTL-5 (9-11)
T/P 5
175(A) 5
175(B) 5
EXT PULSE
LEVEL CTL 6
VCO SELECT-6 (15)
TUNE 6 (8-15)
FM 6
FILL CTL-6 (6-8)
NOISE CTL-6 (0-5)
AM 6
BI-Ø CTL-6 (12-14)
PULSE CTL-6 (9-11)
T/P 6
175(A) 6
175(B) 6
EXT PULSE

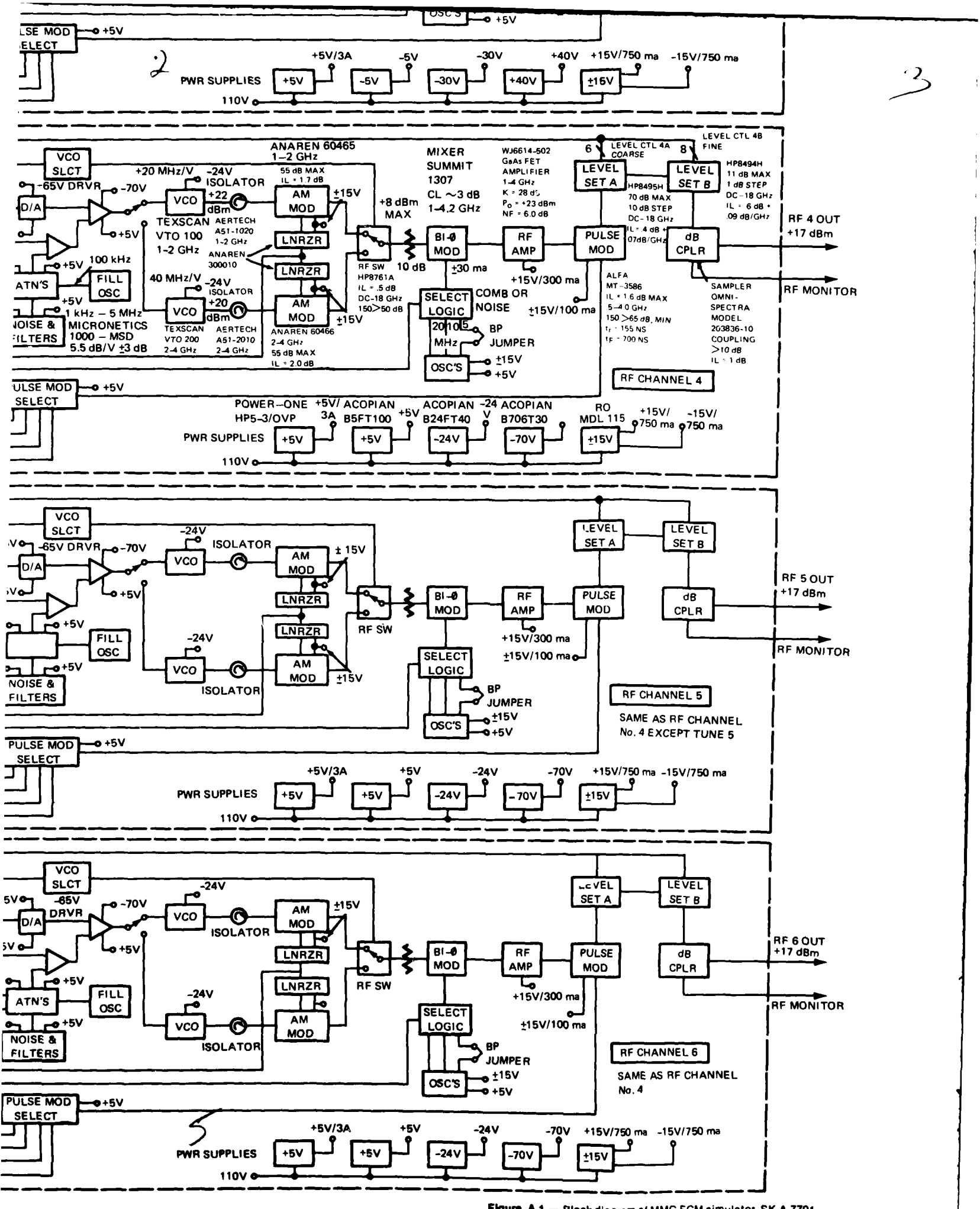


Figure A-1 — Block diagram of MMG ECM simulator, SK-A-7701.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX B

BASIC PROGRAMMING STEPS

The MMG ECM simulator allows a test designer to create ECM outputs by flexibly combining the outputs of various devices in the RF channels and the arbitrary waveform or external generators. The simulator controller can run complex time-varying modulation patterns by timed changes of device settings. Within the limits set by the actual and available simulator hardware and by the speed and memory size of the controller, test designers can write programs for any number of tests. This appendix outlines some basic steps common to any program. The section is necessarily general since there can be many ways of getting the same output.

A test program can be considered conceptually in two parts: the subroutine blocks and the user's program. The subroutine blocks can be used on a black box level to handle the details of a test, such as finding the code numbers, switch settings, addresses, etc. that will give the desired output. The user's program basically handles everything else besides the subroutine blocks, such as specifying the desired output in terms of frequencies, power levels, bandwidths, etc. The user's program determines the form and parameters of a test.

The user's program in turn can be considered in three broad areas: initialization, parameter determination, and output setting. Initialization consists of setting up controller data space, loading necessary files and subroutine block data, and initializing the hardware status. It is the most consistent in form from one test program to another of the three areas. Parameter determination can involve fixed literal numbers, data loaded from tape, programmed algorithms, or operator entries at the time of the test, and may involve a fixed output form or one determined by an operator at the time the test is run. It may be carried out as one stage of a test setup or it may be a repeated part of a cycle. Most of the user programmer's workload will involve parameter determination (especially for complex test forms). Output setting involves the actual setting of simulator devices and running of modulation patterns, primarily carried out when the user's program calls the subroutine blocks. Each area is briefly described below. The reader should be aware that there is some overlap of these areas, such as in the case of a subroutine block that determines each output value in a sequence as that value is needed (e.g., "stepmod"). Parameter determination and output setting implicitly assume that the user has determined the output form (see below).

Initialization sets up the simulator to run a test. This requires that (a) data space be dimensioned in the controller's memory and data loaded from tape storage; (b) any separately taped program segements be added to memory as needed; (c) the simulator hardware be put in a known state; and (d) the state be suitable for the subroutine block operation. The preferred sequence of initialization instructions, with optional steps in parentheses, is:

Dimension data,
(load subroutine blocks),
Call "initial" subroutine,
(enable "shut off" error branch),
Load subroutine data,
(load program data),
(load program segment).

The data that must be dimensioned include those required by the subroutine blocks (see Appendix E), plus any arrays or strings required by the user's program. Simple variables need be dimensioned only if the user intends to record all data. If the subroutine blocks are not an integral part of the program file (i.e., if they are not stored as part of the same tape file as the program doing the initialization), they should be loaded with the load instruction specifying that the program continue at the call part of initialization (see Appendix G). The "initial" subroutine will set the simulator hardware to a desired known state and initialize the array used to hold multiprogrammer card words (see Appendixes E and J). This subroutine should always be called as soon as possible in any program. User programs should also enable the error branch to the "shutoff" subroutine, which will control the program halt that results if the controller (as opposed to the subroutine blocks) detects an error (such as division by zero with flag 14 unset). This enable is not required but is recommended.

After the call to "initial", the user's program loads the data needed by the subroutine blocks (see Appendix E) and may load any prerecorded data needed by the user's program. If the user's program has been written in the form of several separately taped parts, then the initialization part of the program should load the first (noninitialization) part if it was not taped as part of the initialization. A useful instruction that the user can insert after the last tape access instruction is to rewind the tape because this will reduce tape stress and wear if the operator should remove the tape without using the rewind key (see Appendix G). Where the last tape-access instruction is depends on what tape uses there are in the user's program.

Parameter determination and output setting will depend strongly on the user's program. From an output point of view, the user's program is tasked with building up the form of the output modulation; parameter determination and output settings are the means to that end. Determining a test's form can determine fairly well the output setting instructions, and the parameter determination method can be anything that will get the necessary data without conflicting with the output form (this amounts to restricting operator entries during a timed sequence).

Building up a test form requires the user to conceptually break up the overall output pattern into a number of time intervals. Each interval corresponds to a change in the simulator status. The output pattern during each interval becomes that interval's modulation form, and the overall output modulation is a sequence of modulation forms. Commonly, there will be one interval in which the simulator is brought up from initialization to some desired output state in which all modulations (including those run through the controller) will continue until the operator takes some action to change the status (such as turning off the power). This sort of interval, which remains until an outside event interferes, can be termed a stable form and the resulting test a stable test. Its counterpart arises when the simulator is brought up to one output state for a time and then is moved by the user's program to a different state; each interval can be termed a time-varying form and the resulting test a time-varying test.

The concept of an interval may be extended to cover other arrangements in which the program performs a series of related operations, even when the program itself does not control the order or timing of the intervals. For example, if the program prompts an operator to make a number of entries, those entries can be collected to form an operator input interval. If a program is organized as a series of single intervals, separated in such way that the operator controls the sequence of or time between intervals, that program could be considered a multiple interval one.

Within each interval, the user must be able to express the desired output in terms of its parts, where each part can be set up or carried out by a particular subroutine block. The user may provide new subroutines to carry out some output procedure if the existing subroutines are not sufficient. Only those parts that are not continued from the previous interval need be set or run.

The user must distinguish between output parts that are set and those that are run. Both types must be initiated by the controller. The distinction arises in the output behavior when the controller moves to a different task. Set type outputs are stable in the sense that their maintenance does not require use of the controller. Once set, such outputs remain until the controller acts to change or remove that output or until power is turned off. Run type outputs require continuous use of the controller and hence will tie up the controller. Such outputs will remain only as long as the controller continues to run them and will end when the controller moves to another task.

A brief example will help clarify this point. An FM sweep derived from the arbitrary waveform generator is a set type output. Once the appropriate subroutine block has been used to set up the sweep, the waveform generator and auxiliary switch matrix will remain set until the controller acts to change something. Until then, it does not matter what the controller does; the controller is not needed to maintain that FM sweep modulation. On the other hand, an FM sweep derived from the controller through one of the subroutine blocks is a run type output. The controller would be tied up running such an output, and the output would end when the controller moved on to its next task.

Run type outputs can be seen as those that require a dedicated controller to run a rapidly timed series of changes of simulator devices. The devices and changes are the same as for set type outputs, but the timing of individual changes is rapid compared to interval times. In the controller run FM example, the FM sweep is achieved by having the controller rapidly change the tune frequency. If the controller were stopped from the keyboard while running such a sweep, the last tune frequency set would remain set until the program was enabled to continue.

The distinction between set and run (or set and tie-up) is important when the user comes to specifying the order of parts within an interval. Since the controller is tied up by run type outputs, the user's program must carry out any needed set type instructions before reaching a run type instruction in that interval (e.g., to run a synchronous hopping noise spot, the user's program sets the noise spots, carriers, and power levels before running the hop).

The distinction between set and run is also useful when conceptually breaking up an overall output into intervals. Since an interval corresponds to a change in the simulator's status and a run type output modulation will end when the controller stops running it, any run type outputs in a program serve as natural interval markers. The distinction is also used when deciding the parts with an interval. If in the previous interval a modulation has been set, it remains present and need be explicit in a following interval only if some change in the modulation is due in that interval. If a run type modulation output has been run, it will have ended at the end of the interval in which it was run and hence is no longer present in following intervals.

Run type outputs (and any other tasks that will similarly tie up the controller) require some running time control if the controller is not to be completely tied up. Such control for the subroutine blocks is commonly achieved by specifying the number of output steps at a specified rate, e.g., the number of sweeps for the controller's FM sweep, or the number of AM changes for the controller's AM. The time then spent with the controller tied up is the quotient of the number of steps divided by the rate (the number of steps can be specified as the integer product of the rate and tie-up time). In the case of one subroutine block ("stepmod"), the rate may vary from one step to the next so that the tie-up time is implicit in the specified number of steps, while in another ("special") the tie-up time is passed directly. The subroutine blocks will use such information to determine the dwell time at each output step, allowing for the time it takes the controller to execute the instructions that make up that step.

The user can accurately control the amount of time the controller is tied up and so control interval timing. In cases where a run type output should appear as if it were set to run forever, the user can specify some very large number of steps relative to the rate (or a long time if time is passed).

The user may control the time spent on various intervals in other, more general ways that do not involve the number of output steps in a run type subroutine block. In some tests, the desired output may be a sequence of set modulations with no run type outputs appearing. The most direct way of timing control is to use the controller's wait instruction. Waits longer than the 32.767 s maximum of the wait instruction can be reached by repeating several waits or by running a wait loop. The user may also access the multiprogrammer's real-time clock for either relative or absolute timing.

How the user outlines a test program can now be summarized. The user must first describe the desired test output form. This can be done by sketching out the desired output for each RF channel, including time changes (on the level of "a blinking noise spot on one channel, a swept comb on another, the comb being replaced by biphase noise after an interval"). The user may allow an operator to choose the actual form by specifying what part of the user program to run. However, to write a program the user may treat each such operator-selected option as if it were run directly by the controller.

The user must next determine a way of allocating the simulator RF channels; i.e., given the desired output from a channel, the user assigns an actual RF channel as the output source, either by direct (literal) or indirect (variable) assignment. The RF channels are allocated on the basis of the frequency band of the desired output. Indirect assignments allow an operator to specify each actual source at the time the program is used. Only one VCO may be active in an RF channel.

Next, the user analyzes the overall test form in order to conceptually break it up into intervals. The user may find it helpful to visualize or sketch a time behavior chart to indicate RF channel output changes and the relative timing between such changes (see Fig. B-1). The origin of the time axis would correspond to initialization. There will be a minimum of at least one interval in any test; there is no maximum.

Each interval represents a notable change in the simulator status. An interval may involve setting or running some modulation or it may involve no output changes but be used to time an upcoming change, run some lengthy internal calculation, or handle tape access. Since an interval marks a change, only those set devices that change status during an interval need be explicitly present in that interval's instructions.

Having determined the intervals, the user considers each RF channel separately during that interval. If there are no set changes or run patterns involving that RF channel, it can be left as is. If there are changes or patterns, the user must break the desired result into its parts. Any part that was set to its de-

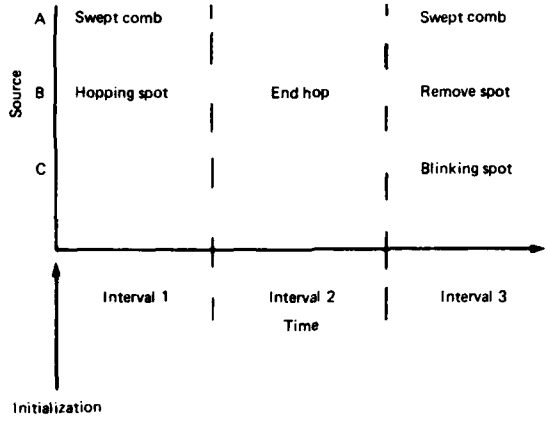


Figure B-1 — Interval forms.

sired form during a previous interval can be left as is. All other parts must be set to their new form and run patterns started (set type outputs are always handled before a run type output in the same interval). Determining the parts of each interval's form is equivalent to determining the output setting instructions.

Having determined the overall form and having broken that into intervals and parts, the user should know from the parts just what parameters are needed. The user must then decide how to determine the actual parameters to be used (e.g., having decided that at some point the output should have a new noise spot, the user determines what RF spot bandwidth and video filter to use). This may range from use of direct literal numbers to operator entries just before each output instruction. When using operator entries, it is generally easier to have the controller prompt for all inputs at one time. The user should obviously avoid operator entries during time-crucial periods such as the middle of a run type output. The user may set up a prerecorded data array containing the usual parameters for a test and allow an interval just after initialization for the operator to pass on any parameters to be changed from the usual or default values.

With the output setting instructions and a parameter determination method decided, the user has the basic outline of a test program intact. The necessary initialization instructions should be quite straightforward and can simply be inserted at the start of the user's program. Table B-1 outlines the general structure of a program.

Table B-1 – New user's outline.

```
Dimension
Initialize
Load data
For each interval:
   Determine interval
   Determine status of general devices (W175s, timer/
      pacer card, external generators; this is usually
      implicit)
   For each RF channel/group of RF channels:
      If active: fix status of set/type devices (VCO select,
      noise spots, biphase modulations, etc.)
      (Only changes from previous interval need to be
      explicit.)
      Else skip
   Next RF channel/group of RF channels
If other tasks (e.g., file loads, printing, checking):
      perform task
If run type routines used:   run
Timing control
Next interval
```

There are a number of additional minor areas that a user should cover besides the major areas of initialization, parameter determination, and output setting. Three of these are: status tracking, test self-documentation, and label use.

Status tracking is usually implicit in the sequence of output settings, which is well-defined when a known, fixed interval sequence is used. Some test programs, however, will allow an operator to determine the order of intervals, as is the case in a demonstration program that through use of the controller's user-defined function keys allows an operator to randomly combine different modulations. In such cases the user's program will need explicit status tracking. Such tracking would typically involve checking if carriers are turned on, if the desired VCO in each RF channel is in use, and (especially) if there is a conflict between using an arbitrary waveform generator as a pulse source or an FM or AM source. If one of the waveform generators has been set up for FM or AM, it should not be switched into an RF channel's pulse circuit, since the waveforms and voltages are not compatible. Status tracking can be done by monitoring the contents of Z[*](see Appendix E) and by use of the controller's flags or a dedicated array.

Test self-documentation can be a very useful part of the overall documentation (see Appendixes D and I). Typically it would consist of using the controller's internal printer to make a hard copy record of the parameter values used. This is particularly useful

when an operator enters parameter values from the keyboard; the operator entries can be printed as entered, which makes it easier for the operator to spot any misentered values. Additional information, such as a message indicating the test form in use, can also be printed if the user wishes. The self-documentation on the controller's printer can be removed and kept in an operator's log or notebook as a direct record of the simulator's status during a test.

Label use can make a test program much easier to understand, debug, modify, or use. Suitable labels can be used to delineate the program tasks of each interval and to indicate the purpose. The user ought to include a good label on line 0 of any program to serve as an identifier. Labels may also be used to tag locations from which an operator can continue a program after a fault-caused crash (see Appendix F). On the HP9825 controller, labels may be up to 70 characters, not counting the colon or apostrophe marks, so there is no trouble with forming easy to understand mnemonics (though shorter labels are obviously more memory efficient than long ones). The only really notable restriction on labels is that each must be unique and the user must avoid using any of the labels already in use by the subroutine blocks (see Table 3).

Labels have an obvious usefulness in handling program branches. Of the branching instructions available on the HP9825, the user should avoid the absolute go to (gto line #) because it makes it awkward to modify a program by adding or deleting lines or to reorder the program sequence. The relative go to (gto # of lines) would generally be avoided because it is easy to drop plus signs when typing that instruction, converting it to an absolute go to and increasing any debugging workload. The relative go to may be useful in rare cases where its slight speed advantage over the jump instruction is needed. Normally, the user will use the jump and labeled go to (gto "label") instructions to handle branching. The jump instruction is space-efficient and allows variables or expressions as its argument. Any jump of more than about seven program lines should be replaced by a labeled go to because longer jumps are hard to follow when reading or debugging a program and may be missed if the program is modified by adding or deleting a few lines. The label used in such a replacement can be any arbitrary symbol (an example is in the "enter" subroutine block). Labeled go to's (gto "label") with good mnemonics should be used when branching between intervals or other main program elements.

There is one other general suggestion that the user may wish to follow when preparing programs, particularly long multi-interval ones. A program may be organized so that the main program consists basically of subroutine calls (a somewhat similar arrangement would be a series of labeled go to's with returns to some instruction that determines the next label). Subroutines may be used even if the subroutine is only used once. The advantages of this approach are better readability, ease of modification, and portability. It can be easier to read and understand the program when the basic instructions (the subroutine calls) can be collected in one group of program lines, especially if good mnemonics are used for the subroutine name labels. Organizing the program as a group of subroutine calls makes it easier to add or delete lines from one procedure without affecting the program flow from one procedure to the next; fewer jump branches need be modified (a procedure is a group of instructions to perform some particular task). A subroutine-based structure also makes it easier to use a particular procedure in other programs by simply using the subroutine that performs that procedure. Such transfers are made easier if the subroutines are written using local p-number variables rather than global variables and if good mnemonic name labels are used (see Appendix D for more details).

It is hoped that the reader will have gathered the general principles of preparing a test program, which should remain valid through simulator upgrades and software changes. Presently the simulator is set up so that it is a one-way device, from controller to simulator to test darkroom. If a look-through/self-measurement or responsive capacity is added to the simulator, the same sort of interval structure can be used; one interval could carry out a look-through and the controller could then responsively decide which output interval should be next, based on what was measured.

How easy or difficult it is to prepare a test program will depend on the desired test and on the user's experience with using the simulator. Before starting a test program from the beginning, the user should check previously prepared demonstration and test programs to see if any can be used as they are, in combination, or with slight modifications, to get the desired output. A substantial library of test programs will be developed for the user to consider.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX C

SUBROUTINE PARAMETER PASSING

To use a subroutine properly, several conditions must be met: it must be the right subroutine for the desired effect; it must be called at the appropriate time and place in the user's program; it must be given the right parameter values for the desired effect; and the parameter values must be in the right form.

The determination of desired effects, timing, and parameter values is up to the user and will depend on the test (see Appendix B). This appendix describes how the user passes parameters to a subroutine. It includes a number of rules and suggestions that particularly apply to those tests in which the user allows an operator to specify the values of the parameters to be passed. The principles of this appendix apply to any subroutine the user writes, as well as to the subroutine blocks.

The user (the one who writes new test programs) must be familiar with the HP9825 controller's HPL language, whereas the operator (the one who runs the test program) need not be (the user and operator will not necessarily be the same person). In passing parameters, the user must follow some well-defined rules concerning the form and order of the parameters; the operator, however, can be more flexible in entering parameter values if the user has provided the appropriate programming. Such programming will increase the workload of the user but greatly decrease that of the operator, when operator entries are used.

Subroutine parameters must be passed in a rigorously defined order. If a subroutine is supposed to be called with duty cycle and period, the user must give the parameters in that order, not as period and duty cycle. The parameter order for each subroutine block can be found in the subroutine descriptions in Appendixes N and O. When the user writes new subroutines, it is up to the user to keep track of the parameters and their order, especially if the new subroutines are to be used in other test programs. This is easier to do if the subroutines use a consistent approach in defining the order of the parameters, such as using the first parameter to specify either the VCO or the band in a subroutine that affects only one VCO or one band.

The user cannot pass characters, strings, or entire arrays to a subroutine. Passed parameters must be literal numbers, simple variables, r-numbers, or array elements; p-numbers can be passed from one subroutine to another. Passed parameters may also be expressions, a particularly useful feature. Scaled numbers cannot be passed as such but must be recast as an unscaled number or an expression. For example, the user cannot pass a parameter value of 5 MHz by writing 5 MHz, but he may pass it as 5M if the simple variable M has been assigned the value of one million. The figure of 5 MHz could also be passed as 5e6 or 5000000 or by passing any variable or expression that has that value.

The parameters used within a subroutine will have a scale that the user must be aware of in passing parameters; e.g., if a subroutine requires a parameter to be in milliseconds, the user must pass milliseconds, not seconds. The subroutine blocks generally treat passed parameters as unit-scaled values. Frequencies and bandwidths are given in hertz, rates in hertz or steps/second, and attenuations in decibels. An exception is that periods are specified in milliseconds while running or tie-up times are in seconds. Frequencies and frequency-related parameters such as bandwidths or rates are given in hertz rather than megahertz or gigahertz to remain consistent regardless of the particular VCO band involved. Tune frequencies would normally be expressed in megahertz in a type I RF channel and in gigahertz in a type II channel. However, by requiring both to be passed in hertz, the subroutine blocks involved in setting tune frequencies ("fset", "fval#") can use the same instructions for both without requiring the user to remember to scale one group of frequencies and not the other. Bandwidths (especially noise spot bandwidths) would normally be expressed in megahertz, but they are passed in hertz to remain consistent with the frequency scale. Rates are given in hertz for the same reason. Since the subroutine blocks use a unit scale for all parameters (except periods), the user can assume the same scale for all related parameters. The user should employ a similar scaling rationale when writing new subroutines, especially those used in more than one test program.

It should be noted that an operator's entries do not have to have the same scale as that required by the passed parameters. The user's program may allow entries in whatever units seem most appropri-

ate for an operator to use and then scale those entries before passing them to the subroutines. This is covered in more detail below.

The required forms for other subroutine block parameters are fairly obvious, with attenuations in decibels, step rates in steps/second, and so on. The arbitrary waveform generator block rates and function numbers can best be understood in the context of using the W175 generators (see Appendix H). Switch codes are used for the pulse and biphase circuits and the FM and AM modulation switch matrices because no alphanumeric can be directly passed; the codes are the actual codes sent to the necessary switches, though this is not an essential feature (see Table C-1).

One parameter form that may lead to questions is that of the VCO number (see Tables C-2 and C-3). Most subroutine blocks affect one VCO, with the VCO indicated by the first parameter passed (the band number serves a similar purpose in subroutines that affect all the VCOs in one band). The VCO number is a short way of specifying which VCO is affected. This information is used in determining the proper multiprogrammer card addresses for the RF channel involved. It is also used in frequency-dependent subroutine blocks to find the right data tables, check parameter limits, and otherwise distinguish between the two VCOs in each RF channel. A single VCO number is easier to use than separate numbers for the RF channel and the VCO in that RF channel. The VCO number of each VCO can be remembered easily by noting each VCOs position in the standard rack arrangement (see Table C-2). It may be noted that several of the subroutines could as readily be specified by the RF channel number. The VCO number is used throughout in order to remain consistent; the user does not need to remember if a subroutine block uses a VCO number or RF number but can just remember the former. However, the user will have to remember if a subroutine uses a VCO number or band number; this can be done by remembering if the subroutine affects one VCO or all of those in a band. The user should also remember that while the VCO number (or band number) is normally the first passed parameter, the arrangement differs in those subroutine blocks that may affect a variable number of VCOs in different bands (such as "DC 175," "T/P"). In these cases, the VCOs are the last parameters passed. The user should use a similar approach in writing new subroutines (see Appendix D). It should be noted that an operator may specify a VCO by RF channel number and VCO letter if the subroutine "enter" is used.

There are a number of methods by which the user's program may find values for the passed parameters. Any of these methods may be combined in a program. The most direct way of specifying values is to use literal numbers. This method is suitable when the test parameter values do not change between test runs. However, literal numbers are tedious to change if the user wishes to vary the values because the user must then rewrite every line with a changed parameter value.

The more usual methods of passing parameter values use variables in the actual subroutine calls, with the methods differing in how they assign values to those variables. It may be noted that if the user considers it easier to specify a parameter in a form different from that required by the subroutine (such as specifying a blinking signal's rate rather than period), the necessary conversion expression can itself be passed as part of the subroutine call, or the expression may be carried out before the call with the result assigned to the variable being passed. The user's choice in such cases can be decided by considering which form would be easier to read, subject to

Table C-1 – Switch codes.

Code No.	*pulse*	*biph*	*auxmod**	*AMaux*
0	Carrier on	20 MHz comb	Off	Off
1	10 Hz, 50% sq.	10 MHz comb	External	External
2	100 Hz, 50% sq.	5 MHz comb	D/A FM	D/A AM
3	W175 (A)	Carrier on	W175 (B)	W175 (A)
4	W175 (B)	40 MHz biphase	–	–
5	T/P card	20 MHz biphase	–	–
6	External	10 MHz biphase	–	–
7	Carrier off	Carrier off	–	–

* auxmod is the FM auxiliary modulation switch control.

Table C-2 – VCO number information.

VCO No.	*Band No.*	*RF No.*	*Letter Code*	F_{min}	F_{max}
1	0	1	A	250 MHz	500 MHz
2	1	1	B	500 MHz	1 GHz
3	0	2	A	250 MHz	500 MHz
4	1	2	B	500 MHz	1 GHz
5	0	3	A	250 MHz	500 MHz
6	1	3	B	500 MHz	1 GHz
7	2	4	A	1G Hz	2 GHz
8	3	4	B	2 GHz	4 GHz
9	2	5	A	1 GHz	2 GHz
10	3	5	B	2 GHz	4 GHz
11	2	6	A	1 GHz	2 GHz
12	3	6	B	2 GHz	4 GHz

the constraint that an expression passed in a subroutine call must be brief enough so that the subroutine's parameter list will fit on one program line (there are no continuation lines for parameter lists in HPL).

Variable assignments may be carried out as part of the program or it may be done in advance. Assignments done as part of the program are somewhat similar to use of literal numbers; however, by collecting all assignment instructions into one part of the program (generally just after initialization), it becomes easier for the user to change values than if the user had literal numbers scattered throughout the program. This approach is suitable for cases in which a test is usually run with the same parameter values, with only occasional changes.

A more general approach is to assign values to variables in advance of a test. The variables are then saved on tape, with the user's program loading the variable data as part of initialization (see Appendix B). By preparing several such taped data files and allowing an operator to select which one is used, the user can provide a rapid and convenient way of changing the test parameter values. This approach is appropriate when a test is usually run with the same values; multiple data files allow easy changes between sets of values. A separately assigned data file can also save program memory space by carrying out the assignments outside the program proper and be conceptually easier to understand. When using this approach, the user is urged to use an array (or strings or the r-numbers) for the taped variables and avoid using the simple variables. If simple variables are used for the taped variables, the variable assignments will be thrown off if the test program dimensioned the simple variables in a different order than was used when the variables were assigned (see the HP9825 reference manuals). If simple variables are used to hold taped data, then the simple variables should be explicitly dimensioned in the same order for both the test program and the assignment of variables. The user must also beware that if ordinary variables are taped and then loaded into r-numbers (or vice versa), the order will be reversed (see the HP9825 reference manuals). The r-numbers are useful for variable length data files since they do not need to be dimensioned.

Table C-3 – VCO number algorithm values.

VCO No.	*RF No.*	*Band No.*	*Band Mult.*	*Tune Slot No.*	*Low/ High*
1	1	0	1	2	1
2	1	1	2	2	1
3	2	0	1	2	2
4	2	1	2	2	0
5	3	0	1	3	1
6	3	1	2	3	1
7	4	2	4	3	2
8	4	3	8	3	0
9	5	2	4	4	1
10	5	3	8	4	1
11	6	2	4	4	2
12	6	3	8	4	0

VCO No.: X
RF No.: int (X/2 + 0.5)
Low/high: int (X mod 4/2 + 0.5)
Band No.: 1 + 2((X - 1)/6) - X mod 2
Band mult.: 2↑(band No.)
Tune slot No.: 2 + int((X - 0.5)/4)
Note: RF function control word slot No. = RF No. + 4

The user's program may also allow an operator to specify data values. This approach is the most appropriate whenever value changes are anticipated more often than occasionally. For operator entries there are a number of suggestions that fall in four categories: prompting, default values, checking, and tracking.

Prompting the operator can identify what variable is involved and the allowable form and range of the variable value. Prompts are closely tied to the entry scheme selected by the user. Entries almost always would be made with a simple one entry per prompt scheme, since this will be both the easiest to write and the easiest to use, expecially for a new operator. Other entry schemes (such as a coded string entry, allowing multiple entries in any order at the same time) may be found useful in certain cases, but generally the operator will make one entry at a time, guided by prompt messages.

It should be noted that, for very long entry lists, the operator need not go through each entry. The user may insert option questions, allowing the operator to end the prompted inputs when the question is reached; the user could also set up the prompts so that a certain reply will end the inputs whenever that reply is made.

Prompted inputs typically would all be made at one time during a test program, after initialization and before the actual output began. During such an input interval, the program would collect all the data needed for the output intervals; the test would then proceed without requiring further operator attention. Inputs might also be made between intervals if the time between intervals is not fixed. The only major restriction on prompted inputs is that in order to avoid timing problems, prompting should not be done during a timed output run or during the middle of an interval.

Prompts should involve at least an enter statement message identifying what sort of value should be entered. Bare prompts (those with no message) should be used only when preparing taped data in advance of a test, and then only when it can be assumed that the user/operator knows the program well enough to identify the purpose of each variable. It should be noted that enter statement messages cannot contain variable display elements but must be a fixed message. The user may use conditional enters, each with a different message, if it seems appropriate.

More commonly, variable elements of a prompt message would be given to the operator by using the display statement just before the enter statement. Any number of display messages can be used for a single prompt, though it is generally best to use the shortest message that will convey the desired meaning; long messages will become tedious, especially after the first time an operator runs the test. When the display statement is used in this context, the time spent on each display should be kept short (a message that seems too fast on the first run of a test will seem much slower on later runs). The stop instruction could be used instead of a wait and the operator instructed to press the CONTINUE key to self-time the messages; however, this may confuse the operator into unintentionally pressing CONTINUE at the enter statement as well as the displays. This procedure is probably better suited to dedicated help programs (covered below). Displays are normally preferred to the printer for prompt messages because the printer is slower, uses paper, breaks up the form of any continuing self-documentation (see below and Appendix I), and soon becomes very tedious, especially for repetitious prompt messages. As an example of a variable

element in a prompt message, suppose the user's program prompts the operator to enter a frequency. A display statement would show the frequency limits (which would vary depending on which VCO is used), while the enter statement message would prompt for the frequency value.

Prompt messages should be kept as simple as possible to avoid slowing down the speed and increasing the size of a program. Ideally a prompt message would fit within the HP9825's 1-line 32 character display. Some abbreviation is useful if it is not overdone. Despite the display size limitations, it is possible to fit satisfactory operator information into the enter message alone so that display messages are often not needed.

It was mentioned earlier in this appendix that the operator is allowed to enter scaled numbers, with the program converting that entered number to the unscaled form required by the subroutine blocks. Thus, in entering a frequency, the operator can specify it in units of megahertz if that is convenient. If a probable scale is known in advance (for example, as it would be for a data calibration program), the operator can be prompted to enter a number directly in that scale (e.g., 5 for 5 MHz). The user may wish to provide corrections for obviously miskeyed entries in this sort of input (e.g., if the operator were prompted to enter a megahertz frequency measurement while calibrating the 500 MHz to 1 GHz band and a value of 1 were entered, the program might conclude that the operator meant to enter 1 GHz or 1000 MHz, and the program could rescale the entered value accordingly).

More generally, the user's program would prompt the operator to enter a value with the scale factor attached (e.g., 3.2g for 3.2 GHz). The user's program would accept the operator's value by entering it to U$ and then using the "enter" subroutine function block to convert that value to a unit scaled form (see Appendix O), the return from "enter" being used to set up the variable actually passed to the other subroutine blocks.

The "enter" subroutine (not to be confused with the enter statement) would be part of the normal handling of operator inputs. It is therefore important to advise the operator that the subroutine will not support expressions sent through the CONTINUE key, unlike entries sent directly to a variable rather than to U$. If expressions are used, they must be evaluated by using EXECUTE before CONTINUE, with the expression either being unit-scaled or the scale indicator added with the edit keys after EXECUTE.

Also, when using the "enter" subroutine, the user should clear U$ before the enter statement to ensure that only the operator's response is in U$ when "enter" is used (U$ is used by several other subroutines; see Appendixes E and O). U$ inputs should be terminated to avoid string errors; for numerical inputs using "enter," the full length of the display is used. The basic form is:

```
" " → U$; ent "message", U$ [1,32]
'enter' →(variable)
```

U$ can also be used for non-numeric operator entries. The operator could be asked to specify a branch or to answer a yes/no question, and the reply evaluated as a string. In most such cases, only the first reply character is significant and need be examined. For example, to ask the operator a yes/no type question, the basic form is:

```
" " → U$; ent "message", U$ [1,1]
if cap(U$[1,1]) = "Y"; gto "yes"
"not yes":
```

Default values should be considered an essential part of an operator prompt. The controller will automatically set flag 13 if CONTINUE is pressed with no data entered at an enter statement and will clear flag 13 if data are entered. The user's program can thus tell if the operator entered anything in reponse to a prompt. By checking flag 13 and assigning a default value to the appropriate variable if it is set, the user's program can allow an operator in effect to skip over an entry, without requiring the operator to enter anything.

This feature is particularly useful when the operator is offered a long list or menu and the operator only wishes to make one or two changes. By pressing CONTINUE at the unwanted prompts, the operator can very rapidly go through a list, pausing only to enter values at prompts for parameters to be changed from some standard value (the operator does not necessarily have to know the defaults). The same effect would be obtained if the passed variables are first set with the standard values (perhaps by using taped data as mentioned above) and flag 13 checked at each prompt to see if any manipulations (such as calling "enter" to get the U$ contents into a variable) are needed.

When the operator makes an entry, the user's program should generally check that the parameter is legal and within range. Such checking often amounts to duplicating some of the checks carried out by the subroutine blocks. It is obviously easier to recover

from a bad parameter if it is found before being passed and causing the system to crash.

There are other checks not carried out by the present subroutine block software that may be needed for some tests. A noise spot may be legally set at one tune frequency but result in a bad output if the tune center is changed or swept. The pulse circuit switch of an RF channel may be set to feed in the signal from one of the arbitrary waveform generators without reference to how that generator is set. There are three functions (FM, AM, pulse) for the two arbitrary generators, so some way is needed to check what each generator is used for (this sort of check is usually implicit in the program but may need to be done explicitly if a program allows the operator to control multiple branching).

Checking can be done implicitly when fixed or default values are used; explicit checks should be carried out on all operator inputs. The logical place to carry out explicit data checks is where the inputs are made; a detected bad parameter can then be rejected while the operator is still present and thinking of that input. Checks would normally be fairly simple and involve testing to see if the value is within some set of fixed or calculated limits (calculated limits allow greater flexibility than fixed ones and can be changed to agree with earlier inputs). If the operator's value is bad, a display message to that effect can be shown, with the program then jumping back to the input prompt. The general form is:

```
dsp "message"; wait (time)
" " → U$; ent "message", U$[1,32]
if flg 13; (default) → variable; jmp (next)
if ('enter' → variable) . . . (within limits); jmp
  (next)
dsp "correction message", wait (time); jmp
  (enter U$)
"next":
```

When checking operator entries, the user may wish to carry out checks other than value limit ones. For instance, if dealing with a very long list, the user's program may designate some symbol to serve as a terminator; each input check would look for that symbol and, if found, go to the next interval after the operator inputs. The user may also include a help file; when the appropriate symbol is found, the program would carry out a dedicated sequence using either the printer or the display to inform and otherwise help the operator.

A help file will be most efficient if taped separately from the program so that it would occupy memory space only when it is needed. Use of a separate help file requires that the user keep explicit track of the line numbers at which the help file and the program should continue, since the present controller has no readily accessible capability to save a program line number (this capability may be added in a future upgrade). Explicit line numbers can make it awkward to modify a program. The user might, in theory, manipulate the error recovery routine in order to save a line number by deliberately causing an error. The recovery routine would have to check if the error were deliberate; if so, the routine would handle the loading of the help file, with recovery being based on the contents of the error line (erl) label. This would still require explicit line numbers within the recovery routine and would increase the size and complexity of a test program and decrease the speed in return for a marginal improvement in ease of debugging a test program. In practice, manipulation of the error recovery routine is not worthwhile.

Help files should be needed only for long, complicated programs intended for use by more than one operator and do not take the place of documenting a test program (see Appendix I). Help files and the number of data checks carried out by the user's program will depend on the user's perception of the operator's skill and needs, the amount of available memory space, and the preparation time available for a test program.

The user's program should keep track of an operator's entries beyond the point where they are passed to the subroutine blocks. Partially this is so that the program can carry out later data checks; it is generally more appropriate in documenting a test run. The user's program should keep track of parameter values at least long enough for the operator to get a record of the values used in the test. The controller's internal printer can be used to record parameter values. Values could be printed as they are entered, which is particularly appropriate when the operator has some control over the next step of the program, since an incorrect entry can then be rapidly spotted and changed. Values could also be saved and all values printed at one part of the program, which is appropriate when the operator makes a few entries in a long list of parameters. The user's program might also save the operator's values on tape for reuse in later runs of the test.

One major principle that the user should remember when preparing for operator inputs is that it is more efficient for the user to define a test than it is for the operator. The user is more familiar with a test program and basically works on it once; the operator works on it each time it is run.

APPENDIX D

USER SUBROUTINES

When preparing a new test program, the user will often want to write a new subroutine for some user-defined purpose. Also, when using certain of the subroutine blocks (see Table 4), the user is required to provide subroutines with a predefined purpose of providing the next value for an output sequence. This appendix gives information for both types. For convenience, the appendix is in two parts: user-defined and pre-defined.

User-Defined

A long or complicated program is easier to read if it is well-structured (a structured program can be loosely defined as one created by the combination of a number of nearly independent building blocks, as contrasted with a program in which every line is sequentially determined by the specific test to be carried out). The MMG ECM simulator software has some structural elements in the subroutine blocks. The user can improve a program's structure through intelligent use of additional subroutines, written by the user. A little structuring effort will pay off in a program that is easier to read and far easier to modify, and that can provide new subroutines for other test programs, saving duplication of effort.

Structuring through subroutines enables the user to organize a test program as a main program and a number of subroutines (which include the subroutine blocks). The main program can then be basically an arrangement of subroutine calls. An advantage of this approach is that it will make it easier to read a program, by outlining in one place what the program does. For example, consider a simple case in which the program prompts the operator through a list of possible inputs, makes a number of calculations using those inputs, and then sets devices and runs patterns. Without subroutines, anyone reading the program would have to read through all the input lines to find that the program does calculations and then read through all the calculation lines to find what the program sets and runs. If more than a few lines are involved, the reader can become confused or lost. By using subroutines, the program lets the reader grasp the program's organization by reading a few adjacent lines, showing that the program calls an input routine, a calculation routine, and, finally, a set and run routine (or routines).

Such structuring can be particularly useful in a multiple-interval test program (see Appendix B). Aside from making it easier to follow the overall form, a subroutine-based structure would make it much easier to modify a test by changing interval order or adding or deleting intervals; the user would only need to manipulate single instructions (the subroutine calls) rather than entire procedures many lines long.

When using subroutines to structure a program, the user should try to use good explanatory label names (see Appendix B). A number of studies have suggested that human short-term memory typically holds six or seven items. Good label names can reduce the number of items a reader must remember in order to understand a program.

The six- or seven-item figure for human short-term memory suggests that, when dealing with a very complex program, the user should try to structure the program so that any one interval (or other program division) could be understood as a sum of six or fewer parts. The user can stack subroutines to this end, having one subroutine call a number of others. The only limit to the depth to which subroutines may be stacked on the HP9825 is the available memory; if memory size does become a problem, the user can break the program into a number of separately taped segments (see Appendix G).

Another advantage to using subroutines is that they can be used more readily to carry out the same task in other test programs. The user faced with a task common to several test programs could write a single subroutine shared by the tests, rather than having to rewrite the task instructions for each test. The user could build up a library of subroutines that join the subroutine blocks in being readily available to each new test program, the new subroutines being saved on tape. To fully exploit this capacity, the user should follow certain rules and suggestions below on variable use.

In the HP9825 HPL language, all variables except subroutine p-numbers are global. These global variables may be read or assigned values anywhere in a program. Local p-numbers are dynamically allocated

when a subroutine is called and are lost when the subroutine returns. The p-numbers (p#s) of one subroutine are independent of the p#s in another subroutine (save when one subroutine calls another, passing p# parameters; an assignment to the p# in the second subroutine corresponding to the passed p# in the first subroutine will affect the value of the p# in both subroutines. See the HP9825 reference manuals).

If a subroutine is to be generally useful in different test programs, it should use p#s, reserved subroutine variables (simple variables U-Z), and general data (X[*], Z[*], etc.; see Appendix E). This requires that all necessary parameters be passed in the subroutine call. While this involves more programming effort than direct use of global variables, it frees the subroutines from dependence on the particular variable used. This in turn gives the program calling the subroutine more flexibility in using the global variables and allows expressions to be used in the calls. For example, rather than writing a subroutine that uses the variable F to hold a frequency value, the user would use a p# (such as p2) for the frequency. The subroutine calling program (which may itself be a subroutine) would have to pass the value, but it would be free to use variables other than F to hold the frequency value passed and could also pass expressions or functions, while F could be used for some other purpose.

A rule of thumb for variable use in such subroutine structured programs is to use the global variables in setting up parameters passed with the subroutine calls, and p#s and reserved data (see Appendix E) in the subroutines. A subroutine should use a global variable other than the reserved variables only in well-defined special cases, such as the nextval subroutines (below). The reserved simple variables are particularly used as for/next loop indices.

Another suggestion for writing subroutines is to use a consistent order in passing parameters (see Appendix B). It will make it easier to use that subroutine if the user can readily remember the parameter order without having to look it up. A consistent order also makes reading and modifying a subroutine easier. In this sense, a consistent order can be considered a logical extension of using good mnemonic label names. A rigorously consistent order is not required and exceptions can be made, but the following can be taken as guidelines:

1. In subroutines affecting one VCO (or all VCOs in a band), give the VCO number (or band number) first. This may be extended to a case using a fixed number of VCOs not necessarily in the same band; give the VCO numbers first.
2. In subroutines affecting a variable number of VCOs, give the VCO numbers last (use p0 to find the number of VCOs after allowing for other passed parameters).
3. Give deviation centers (if applicable) before deviation magnitudes, magnitudes before rates, and rates before miscellaneous parameters (e.g., "swp175" passes VCO number, center frequency, FM bandwidth, W175 block rate, and W175 function number, in that order).
4. Pass parameters as unit-scaled numbers (except periods, which are in milliseconds), to remain consistent with the subroutine blocks (see Appendix C).

The user will probably appreciate that a subroutine structure should make it easier to prepare long programs and to debug and modify shorter ones as well. When preparing a relatively short or simple program, especially one prepared on short notice, the user may find it easier to use a straight-ahead approach with no subroutine structure. This can be perfectly acceptable, and an operator generally would never see any difference. If a short program is wanted for a brief run, it may make no difference to the user either. The user preparing a test should ask the hypothetical question of whether the user expects to modify that program later, or use part of it in a different test, or if a different programmer will be using the program in some way. If the answer is yes, it could be more efficient to use a good structure from the beginning, rather than to try to impose one later.

When actually writing a user-defined subroutine, there are fewer specific rules or suggestions to pass on, since the subroutine form basically depends on what the user wants done. It has already been noted that the user is assumed to be familiar with the HPI language and so should need no reminders about multiple instructions on a line, use of p0, and defaulting passed p#s to zero by not passing them. A number of general suggestions follow.

A user-defined subroutine may or may not involve sending data over the bus. The user should be able to handle most bus-involving data by using the subroutine blocks. When the user does wish to send something over the bus from a new subroutine, one of the existing subroutine blocks can serve as a model for the data formatting, addressing, and tracking (see Appendix K). It is expected that most cases in which the user wants a new subroutine to send something over the bus will involve setting the arbitrary waveform generators or running some complicated multi-

ple VCO output pattern. The former basically involves sending a properly set up string to the right W175 address (see Appendix H) and the latter is basically a matter of finding data, formatting it, and sending it to the right place at the right time. In this latter case, a nextval type approach can be used (see below). The user should base the required return form on an estimate of the highest desired rate (e.g., if a run-type output involves changing tune centers at a low rate, the nextval could return single D/A numbers and could get those numbers from frequencies using "fval#". If a high rate will be needed, the nextval should return tune words taken from a table prepared in advance; see below). Also, in running timed output patterns, the user should allow for the program instruction time in setting waits, as is done in, for instance, "stepmod." The necessary offset times could be held in an expanded X [*] or in some other user-selected variable, with the former preferred when the user subroutine is to be used in other programs.

The user may also include some simple value checks to ensure that passed parameter values are legal (again, this is especially appropriate if the subroutine will be used in other test programs). The sort of tests and the response to any error will depend on the expected future use of the subroutine. If it is used only in a specific test, the subroutine checks may involve the values of known global variables in that test and so allow one subroutine check to involve knowledge of other results. The error response may involve getting the operator to fix the bad condition. A general use subroutine should restrict its checks to the passed parameters and the data contents, so that each subroutine's checking is independent of others. The error response could not assume any knowledge of the calling program's structure and so should halt the simulator through "err stp" (the user should make up a unique Z code; see Appendix F).

To maintain good structure, each user-written subroutine should carry out one primary task. A larger task may be broken into subtasks performed by separate subroutines. Thus, if for example the user wished to fill a number of r-numbers with values and then to send those values to various RF channels, the user would use one subroutine to get the r-numbers and another to run the actual output. This would improve the structure's clarity and make it easier to change the nature of the output values, by using a different subroutine when getting the r number values.

In all user-written subroutines, the final choice of what the subroutine does and how it does it is up to the user. It is up to the user to decide if the additional programming required by subroutine structuring, data checks, and the rest would be repaid by ease of later use and modification.

Pre-Defined

In contrast to the user-defined subroutines are those in which the purpose has been pre-defined but whose form is largely up to the user. These subroutines are termed next value or nextval subroutines. An explanation of their purpose will show how the nextval label is self-defining.

It was mentioned in Appendix B that some run type subroutines tie up the controller in order to run an output pattern, and it was mentioned above that it can improve a program's structure and make modifications easier if each subroutine performs one primary task. The run type subroutines adhere to this concept. The basic run type subroutines (first column of Table 4) are primarily concerned with the proper addressing of a number of output values in a sequence. These subroutines in turn call on other subroutines to get the actual values output. This latter group of subroutines is called on each output step of the run type subroutines to provide the next value; hence as a class they can be called nextval subroutines.

The main advantage of this approach is that it enables the user to specify new run type output forms without having to rewrite the controller-simulator address manipulations each time (indeed, the casual user does not even have to be aware of the internal addressing). The user has only to specify the nextval subroutine to specify the output and can do so in a number of ways, giving the user great flexibility.

This flexible adaptation can be carried out as part of a program. For example, the user might provide a number of nextval forms and let the operator specify which to use in a given test run. The user could also change the nextval during a multi-interval test, so that the same run type subroutine when called in different intervals would result in different outputs.

Nextval subroutines are one exception to the general rule against using general global variables in a subroutine. Conceptually, any nextval subroutine can be regarded as being specific to the test in which it is used, so there is no portability from one test program to another to be lost if the general global variables (see Appendix E) are used. Through such variables the user can have the main program set up or modify the run type output.

The calling subroutines must provide the addressing and basic timing control for the output, as well as any necessary manipulation of the nextval return. The kind of data manipulation required depends on the format of the nextval return, and this, in turn, will be determined by the expected maximum output rate. When a high rate is needed, data manipulations during the output run should be kept to a minimum. If necessary, fully manipulated data can be calculated in advance of the output run; and tabled in an array or as r-numbers; the nextval would then be concerned with finding the right table entry.

The user must pay certain attention to the output timing when writing a nextval subroutine. Any run type output implies that the controller sets the output rate by controlling the time any one output value remains set (the dwell time). Typically the dwell time is implied by a passed rate, but it may itself be a nextval return (e.g., "stepmod"). In setting up a wait instruction to match the desired dwell, allowance should be made for the time required by the actual output instructions (the loop time). This is done by off-setting the dwell time by the loop time to get the wait time. For example, if a desired dwell is 40 ms (corresponding to a 25 Hz output rate) and the output instructions take 35 ms to complete, the actual wait time would be 40-35, or 5 ms.

The user will appreciate that the loop time represents the minimum actual dwell, corresponding to a wait of zero, and hence that the loop time sets the maximum output rate. Some of the instructions in an output loop are essential: the output loop control, the nextval call, the output write over the bus to the multiprogrammer, and the check to allow the multiprogrammer to handle one output before sending the next (see the description of "?", Appendix O; an analog would apply if the output loop did not involve the multiprogrammer but involved some other bus device). The time required for these instructions sets a minimum dwell time restriction on the controller. On the HP9825 this time is about 35 ms, corresponding to a maximum rate of about 28 Hz.

A run type subroutine could generally be written so as to allow the maximum output rate; this can be done by requiring that at least some of any needed data manipulations be carried out outside the calling subroutine. For example, a run type subroutine that sets new tune center frequencies could require that its nextval subroutine return the D/A number to be set. The nextval subroutine could get the D/A number from the next frequency value or it could look up a table of D/A numbers prepared in advance. How the nextval gets the D/A number would make no difference to the calling subroutine, with one exception.

Since the essential instructions in any run type output loop include getting the next value, the loop time must reflect the time needed to get the value. If there are two alternate nextvals for a calling subroutine, one simple (say, a D/A number table lookup) and the other complex (say, a calculated frequency converted through "fval#" to a D/A number), then the loop times can differ significantly and the dwell offset must be adjusted to fit the nextval used.

In order to allow for this, the loop time should be represented in the calling subroutine by a known data variable. If the loop time is then changed by changing the form of the nextval, the value of that variable can be adjusted. The subroutine blocks use entries in X [*] to hold loop times. The user writing new run type/nextval subroutines can expand X [*] to hold the new values, or could assign some other data variable for the same purpose.

Loop times can be estimated after a little practice or measured as the difference between a known wait time set in the program and the observed actual dwell time. Such differences are easier to measure at low rates than high ones. It can also be easier and more accurate to measure actual dwells by observing voltage changes inside an RF channel using an oscilloscope (especially one with storage capability) than by observing RF output changes on a spectrum analyzer, though the latter is easier to hook up.

Additional instructions besides the essential ones may be part of the calling subroutines (or the nextvals), such as data checks and manipulations. While data manipulations as part of a calling subroutine will reduce the maximum output rate, they also make it easier to write the nextval, an advantage if a casual user prepares a nextval because it requires less detailed knowledge and less effort.

The number of individual outputs during a run type pattern is directly or indirectly controlled when the main program calls the calling subroutine (see Appendix B). Typically this is done by setting up a for/next loop in the calling program, using a reserved simple variable (i.e., X) as the index. The nextval may read the index value if the user so wishes (but should not change that value). The tie-up time is the sum over the number of individual outputs of the individual dwells (the dwells may vary if the output rate is not fixed). The run type subroutines set the number of outputs in the for/next loop, with dwells set either fixed by the rate (e.g., "AMown") or by a nextval return (e.g., "stepmod").

The remainder of this appendix describes three subroutine blocks that require the user to provide nextval subroutines: "AMown," "stepmod," and "special." The user will find that a number of details below can be readily adapted to new run type subroutines.

The subroutine block "AMown" shows how to use the digital output cards in the HP6943 extender to get an AM signal (a directly analogous subroutine could be written to use the FM card). The subroutine was originally written before a second W175 waveform generator was added to the simulator as an AM source. The subroutine remains useful since it provides a way of getting AM if the second W175 is preempted for use as a pulse source. The subroutine also has a larger valid range than does the W175 (see Appendix H).

"AMown" is called with a fixed rate and a specified number of output steps. The rate is given as the number of steps per second (see Appendix O). The required nextval subroutine is "AMval." "AMval" has the VCO number passed as p1 and may read the output loop index (the output step number) in X. The nextval must return a value representing decibels of attenuation relative to the current output power level. The decibel value is returned by assigning it to the variable U; its usual range is 0-55 dB.

A simple example of an "AMval" form can be used as a default that will result in a sine squared output:

"AMval": 30|sin(X)|2|→U; ret.

The main program should have set radian angular units. In place of the fixed maximum of 30 dB, the user could use any available simple variable, assigned a value before "AMown" is called, or perhaps an array using the passed VCO number in p1 as the array index.

The "stepmod" subroutine is called with the number of output points. The subroutine allows an arbitrary frequency pattern to be run through the controller. Each output point requires a tune card D/A number from the nextval "stepval" and a dwell time at the resulting frequency from the nextval "stepwt". The loop index is in X and both nextvals are passed the VCO number.

The nextval "stepval" returns a D/A number through U. Any integral number 0-255 will be accepted by "stepmod;" it is up to the user to ensure that the result is meaningful. The nextval "stepval" could calculate a frequency by some algorithm, check that the frequency is legal, and use "fval #" to get the D/A number. It could also get the D/A number directly, including implicit checks in the way it gets the next value. As an example, suppose the user wanted to hop the tune center randomly between two frequency limits. The main program could use "fval #" and the min and max functions to get the lower D/A number limit in L and the difference with the higher D/A number limit in D (the low D/A number does not necessarily correspond to the low frequency). The nextval would be:

"stepval": L + int(Drnd(1))→U.

The nextval "stepwt" returns a dwell time through U. Its range depends on the loop time value in X[2] but should typically be about 40 to 32,807 ms. The user can use a fixed dwell or one that is fixed for a number of output steps (or an amount of overall time) and then changes, or one that varies on every output step. A simple example for a fixed rate might assume that the main program has put a rate value in steps/s or Hz in the variable R:

"stepwt": 1e3/R→U; ret.

The "special"subroutine is called using rate and running time. It is used to synchronously hop all three VCOs in a band. The subroutine was written to allow for high ouput rates, with a measured maximum of about 26.3 Hz. To get the higher rates, the output values must be calculated and tabled in advance. The table length parameter is passed to specify how long the table is and hence how many entries to read before repeating the table.

The r-numbers are usually used to hold the data, rather than an array, since the r-numbers do not need to be dimensioned (this makes it easier to vary the table length). The passed table length does not have to be the full table length; by using only part of the table, the user can examine the effect of the nonrandom repeating of table values. The length of the table should ideally be as long as possible; the user may use this subroutine as a separately taped segment so as to eliminate other parts of the program and so allow more data space. A string array is not suitable for holding "special" values; at high rates it would take too long to convert the string to numeric form.

The subroutine needs five values on each output step. Two of these represent the three tune center values and three represent the RF channel function

control words. By sending the latter out on each output step, the subroutine allows a number of special moves: noise spots can be kept constant even as the tune centers change; the noise spots can be varied after a certain amount of time; and the carriers can be turned off to simulate a look-through. The channel control words could also be left alone for a simple, easier to program version of the output pattern.

The nextval subroutine "valspec" must return the required data in specified variables, as follows:

U - double tune word,
W - single tune word,
Y - channel control word, A,
Z - channel control word, B, and
V - channel control word, C.

The three VCOs in a band can here be given the labels A, B, and C, in order of increasing VCO number (these labels are not related to the VCO A or B indicated on the front panel of the simulator). The data returned from "valspec" is passed on to the multiprogrammer without any checks or manipulations. As a result, the tune data must be already formatted, making this the most difficult nextval for the user to prepare (an easier way of getting hopping can be written, but the maximum rate is only about 12.3 Hz).

The data for this nextval must be prepared in advance. For each output step, the user must have three D/A numbers which may be found by calculating a frequency and finding the D/A number or by directly getting a D/A number (this part is strongly similar to the "stepmod" nextval "stepval"). The numbers are then shifted as necessary, so that a D/A number for the high end of a tune card is shifted eight positions and one for the low end is unshifted (shifted 0). Two of the numbers must then be put together as one word (using inclusive OR).

To remember which VCOs are put together, the user need only note if the VCOs are in the lower bands (a type I RF channel, bands B and C) or the higher bands (a type II RF channel, bands D and E/F). Using the VCO A, B, C notation mentioned above, the necessary shifts and inclusive OR's are indicated, using (s) to indicate a high end shift and / to indicate two numbers put together:

low band: A/B(s),C,
high band: A(s), B/C(s).

If the RF channel function control words are to be actively changed during a "special" run, data values must be set up for them. This can be done by using the appropriate Z[*]entries as a basis. The Z[*] locations can be found from the band number in a manner analogous to that used in the second line of "special." The VCO select, pulse and biphase carriers, and initial noise spots can be set, giving Z[*]a set of initial values (if necessary, the power level can be kept at 81 dB down to keep any output from appearing while the data are set up). Then, for each noise spot (or other) change, the appropriate subroutine "fnoise" (or "pulse" and "biph" to turn carriers on/off for look-through) is called, after which the Z[*]contents are saved. The process could be continued for each output change.

If a simpler approach is used and the RF channel function control words remain fixed for the test run, then the user can make a single direct assignment from the appropriate Z[*] locations to the variables Y, Z, and V.

The nextval subroutine itself would basically be concerned with finding the right table entry for each variable on each output step. The step number or loop index can be read in X and the table size is passed as pl. Typically the nextval would use the step number in X to find a basic table location and would count a fixed number of entries past that basic location to get all the variables for that step, while the table length is used to fold around the end of the table and so repeat. The table length should be some integral multiple of the number of entries used at each step (the for/next loop set up in "special" may need to be modified to reflect the number of variables used on each step).

As an example for "valspec," suppose we have the simple case in which the channel control words remain fixed. In this case the main program would directly assign the appropriate contents of Z[*] to Y, Z, and V (after setting the VCO select, carriers, and noise). The r-numbers would contain the tune words, prepared before the "special" call. The numbers could be prepared as part of the program or in advance and saved on tape. The double word would be held in the even-numbered entries and the single words in the odd-numbered entries. The example for "valspec" would then be:

```
"valspec": r(X mod pl)→U
          r[(X + 1)mod pl]→W; ret.
```

There is an obvious extension of this to cover the case when the channel control words do change, or those words could be held in a separate array or by rearranging the r-numbers to hold contiguous groups of the same variables (i.e., all values for U followed by all values for W, etc.). It is up to the user to decide what nextval form is needed for a particular test.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX E

DATA AND VARIABLES

This appendix describes the data and variables used by the subroutine blocks. As has been mentioned in Appendix D, the subroutine blocks chiefly use local p-numbers as the internal variables. Global variables are used as for/next loop indices, to hold data, set up output strings, handle some value passing (between a run type subroutine and an associated nextval subroutine; see Appendix D), and handle error messages.

The user may freely use all global variables not reserved by the subroutine blocks and may use all of the free-use flags (flags 0-12). The user may also read the data values held in the reserved variables. Table E-1 contains a guide to user assignments involving the reserved variables; a rule of thumb might be for the user to avoid reserved variable assignments except for using U$ in handling operator inputs. For reference, the reserved variables are:

strings: U$, V$, W$, X$, Y$, Z$,
arrays: X[*], Z[*],
simple: U, V, W, X, Y, Z.

Data are used by the subroutine blocks chiefly to determine what values are sent over the bus in order to get a desired output. The main bus messages are directed to the tune and channel function control cards in the multiprogrammer. Proper output word manipulations using the subroutine blocks require changes to the normal (wake-up) state of the controller and of the multiprogrammer. The changes are related to the data forms that enable the subroutine blocks to work independently (see below) and somewhat complicate use of the controller's flag 14.

Table E-1 – User assignment of reserved variables.

Variable	*Assignment Rule*
Z$	Never
X$, Y$, W$	Never (update using "load-$" subroutines)
Z[*]	Extreme caution
X[*]	Caution
Simple variables (U-Z), V$	Assigned values/contents will be overwritten by some subroutine blocks
U$	Use for inputs ("enter"). Assigned contents will be overwritten by some subroutine blocks

The controller wakes up with its binary operations (shift, inclusive OR, etc.) in 2's complement mode. This complicates control of the operand word's most significant bit (bit 15), which is needed in setting tune and function control card values. The shortest and easiest solution is to have the controller's binary operations carried out with flag 14 set, which sets the format to unsigned binary.

On paper, the binary operation result could then be sent to the multiprogrammer, in which the digital output cards wake up in unsigned binary format. In practice, it was found that this would result in a multiprogrammer error when the word sent has bit 15 set (regardless of whether flag 14 was still set or not at the time the word was sent). The multiprogrammer in its wake-up mode will accept a decimal value word with bit 15 set (e.g., 65510) if that word is sent as a literal but not if the word is formed through the controller's binary operations. It seems as if, while the controller will allow a word to be formed as an unsigned binary number when flag 14 is set, that word will be passed to the multiprogrammer as if it were a negative number when bit 15 is set (i.e, it seems to go back to 2's complement regardless of flag 14) and the multiprogrammer cards will refuse to accept that word as an unsigned binary pattern, but will treat the word as an illegal negative number.

It was found that a proper and acceptable data transfer between the controller and the multiprogrammer can be made if the appropriate multiprogrammer digital output cards are put in 2's complement format. The cards in slots 2 through 12 (which handle the tune and channel function control words, and the auxiliary switches and level set attenuators) are put in 2's complement by the "initial" subroutine block.

A straightforward way of setting flag 14 to change the controller's binary operation format would be to set it just once, in "initial," so that flag 14 would always be set in a program. However, when flag 14 is set, the controller also sets a number of defaults for

illegal math operations, such as division by zero, so that illegal math operations will not halt the program. This could give the user and operator a false idea of what the simulator is doing.

In order to avoid this, flag 14 is not set by "initial" but by the subroutine blocks that require the flag to be set. The flag is set only when it is needed and will not lead to unintentional math defaults. The subroutine blocks using flag 14 will clear the flag before they reach their return statements. It is important that the user remember this if the user does want flag 14 set outside of the subroutine blocks. In such a case, the user has two options: The user can simply set flag 14 repeatedly, either whenever it is needed or after calling any subroutine block that used the flag, or the user can set flag 14 once and modify the subroutine blocks by removing the clear flag 14 instructions. The latter is more appropriate when the subroutine blocks are taped as part of the test program and the user is cautioned not to later mistake the modified subroutine blocks for the unmodified ones. As for identifying which subroutine blocks use flag 14, a rule of thumb is that any subroutine block that sends data over the bus will directly or indirectly use the flag (the short descriptions in Appendix O will mention if the subroutine does not use the flag; otherwise the flag is used).

The controller data form allows independent setting of the simulator devices. The Z[*] array is used for this purpose. This array holds replicas of the multiprogrammer card words; i.e., Z[10] contains the word sent to the multiprogrammer card in slot 10 (see Table E-2). When a subroutine block is called on to modify some simulator device controlled by part of a card word, it will modify the appropriate part of the right entry in Z[*] and then send the new Z[*] word to the card. The subroutine block does not have to track or reset the other devices controlled by the same card. For example, the biphase circuit of an RF channel is controlled by bits 12-14 of the channel's function control card; these bits can be set by "biph" without any explicit reference to the VCO select, pulse, or noise settings.

Z[*] is dimensioned and used in preference to some other approach since it allows a good combination of speed, data tracking, and memory efficiency. By keeping replicas of the card words in the controller, the subroutine blocks do not need to read back the card words every time they are to be changed. In case of an error shutdown, the simulator outputs can be removed by sending the appropriate data to the cards while keeping the previous card values intact in Z[*]. The Z[*] contents could be used in debugging the crash, and the output could be restored by simply sending the Z[*] contents back to the cards. The card word approach itself is more efficient (and easier to follow) than direct setting of individual bits.

Table E-2 – Z[*] contents.

No.	*Z[*] Contents (HP6942/3 Card Words)*
1	Memory card word
2	Tune word, RF 1 and 2
3	Tune word, RF 3 and 4
4	Tune word, RF 5 and 6
5	Channel function control word, RF #1
6	Channel function control word, RF #2
7	Channel function control word, RF #3
8	Channel function control word, RF #4
9	Channel function control word, RF #5
10	Channel function control word, RF #6
11	Power amplitude level set
12	Aux. FM/aux. AM switch matrices
13	T/P card, overall pulse width
14	Counter card word
15	D/A AM, RF #1
16	D/A AM, RF #2
17	D/A AM, RF #3
18	D/A AM, RF #4
19	D/A AM, RF #5
20	D/A AM, RF #6
21	D/A FM
22	Digital input card word

Z[*] must be dimensioned before initialization but does not need to be loaded with data as "initial" and the subroutine blocks will handle the contents set up. An exception is that the user may on occasion want to assign a stable nonzero decibel of attenuation value to the AM D&A cards in slots 100-105 (normally the level set attenuators would be used to get such a result, and it is doubtful if a power attenuation greater than the 81 dB to be had from the level set attenuators has any real meaning).

Within a subroutine block, the binary AND operation (band) is used to mask out the unaffected part of a card word while clearing the affected part, shift (shf) is used to line up the new value in the proper bit positions, and inclusive OR (ior) combines the shifted new value with the other card word contents. The result is saved in Z[*] as the new card word.

The run type subroutine blocks differ slightly in that the new card word formed on each output step is not saved in Z[*], saving a little on loop time (band, shf, and ior are otherwise used as in the previous paragraph). When a run type subroutine completes its run, it will (except for "special") send out the prerun Z[*] contents, restoring the original output.

The reserved variables other than Z[*] are used to handle calibration data, controller-tape and controller-W175 waveform generator transfers, inputs, for/next loop indices, nextval returns, and error messages. These uses are discussed below according to the variable used.

Z$[12,54] holds the complete tune frequency calibration data for the simulator. Each numbered string in the string array contains the data for the VCO with the same number (i.e., Z$[8] for VCO number 8, etc.). Each string is organized as six sets of nine characters, each set defining one point of the frequency-D/A number calibration curve. The nine characters of each set are organized as five frequency characters followed by four D/A number characters.

The strings are arranged so that the tabled frequency increases as the string index increases. The D/A numbers may be increasing or decreasing. The lowest and highest (first and last) tabled frequencies are taken by "fval#" as the limits for legal frequencies from that VCO number. Unlike the frequency checks based on a band number calculated from the VCO number, this check is independent of the actual arrangement of VCOs in the simulator rack and depends only on the data put in Z$. The Z$ data limits could also be set to represent the real-world limits of each VCO tuning curve rather than the paper specifications.

Frequencies are tabled in Z$ as fractional numbers in gigahertz units ("fval#" will scale the number taken from the table to hertz units). With the leading character space blank (a result of forming Z$ with the controller's numeric-string conversion operations), there are four significant frequency characters, including any decimal point. For example, 250 MHz is tabled as 0.25 with a leading blank. D/A numbers are tabled as integers in the range 0-255. The D/A number used is found by linear interpolation from the tabled data.

Z$ is loaded after dimensioning, from file 2 on track 1 of a standard tape. Once loaded, it needs no updates during a test run. The user may read values from Z$ but should never assign to it because that would overwrite the data contents.

X$[6,120] and Y$[6,120] hold noise generator and fill oscillator calibration data, respectively. To save memory space, only partial data sets are held in the controller at any one time. Each individual string contains data for the active VCO in the RF channel of the same number (i.e., Y$[4] for RF channel number 4, etc.). X$ is further constrained in that the data contained are also those for the video bandwidth in current use.

The noise calibration (noise generator and fill oscillator) is organized on hold points on the bandwidth-attenuator setting calibration curves. The attenuator setting (from 0, no attenuation, to 7, full attenuation) is implicit in the position in the string of the corresponding bandwidth value. The bandwidth that results from a given attenuator setting will vary with the band position of the tune center about which the bandwidth is measured. The noise data are calibrated at three positions throughout each VCOs frequency band to get data for low, mid, and high frequencies. The tune center determines the part of the band in use and hence which set of data should be used. This information can be indicated by a band part number: 0 for low band, 1 for mid band, and 2 for high band. Each band part covers one-third of the VCOs band. The band part is used as an index offset in finding the right part of X$ or Y$.

The individual strings of X$ and Y$ are organized as three sets of 40 characters. Each set contains the data for one band part, with increasing string index corresponding to increasing band part (increasing frequency). Each set is organized as eight groups of five characters. The characters contain the bandwidth values in order of increasing attenuation. The attenuator setting used is found by finding the closest tabled frequency to the desired frequency.

Frequency values in X$ and Y$ are similar to those in Z$ except that the noise bandwidths are tabled in megahertz units ("fnoise" will scale the numbers taken from the strings to hertz units). All tabled bandwidths should be measured by the same standards, i.e., 10 dB down from peak (any other dB down level could be used as long as its use is consistent).

X$ and Y$ can be loaded after dimensioning from files 6 and 5 on track 1 of a standard tape. When loaded from those files, the noise strings will contain data for the B labeled VCOs in each RF channel (i.e., the higher frequency VCO in each head), with X$ containing the data for the 5 MHz video noise bandwidth (video number 5; see Table E-3). During a program, the user may change the string contents to reflect a change of the active VCO in an RF channel or a change in the video filter by using "loadX$" and/or "loadY$."

It should be noted that the current subroutine block software does not track which set of data is actually in X$ and Y$. It is up to the user to do this and to make any necessary "load$" calls. If "fnoise" is called with incorrect data in X$ or Y$, those incorrect data will be used, giving erroneous results. It should also be noted that calling "fnoise" with a video number of zero, which turns off the noise generator and leaves the fill oscillator on, does not actually involve any use of X$. Thus, a video number change involving a video number of zero does not require the user to update X$ and can be effectively ignored (i.e., going from a video number 5 to video number 0 to video number 5 is effectively no change as far as the contents of X$ are concerned; going from video number 5 to video number 0 to video number 4 is effectively going from number 5 to number 4 and does require a change in the X$ contents).

The user may read values from X$ or Y$ but should never assign to them.

W$[6,120] holds W175 arbitrary waveform generator FM calibration data. Its organization and use are similar to those of Y$, except that the string positions of the bandwidth values correspond to W175 voltages. The voltage that corresponds to a particular bandwidth entry is found from the bandwidth's position and conversion factors found in X[*] (see Table E-4). The voltage samples are equally spaced. The voltage sent to the FM W175 is determined by linear interpolation from the tabled data.

The contents of W$ may be updated using "loadW$" when the active VCO in an RF channel changes and the W175 is used for FM. The user may read the contents of W$ but should never assign to it.

Table E-3 – Video filter numbers and bandwidths.

Video No.	*Video Bandwidth*
0	Off
1	1 kHz
2	10 kHz
3	100 kHz
4	1 MHz
5	5 MHz

Table E-4 – X[*] contents.

No.	*X[*] Contents*	*Usual Value**
1	"special" loop time (ms)	36
2	"stepmod" loop time (ms)	40
3	"ownswp" loop time (ms)	35
4	"AMown" loop time (ms)	40
5	Fraction of spot RF BW from fill	0.2
6	Max. output power, 250-500 MHz VCOs (dBm)	16.5
7	Max. output power, 0.500-1 GHz VCOs (dBm)	16.5
8	Max. output power, 1-2 GHz VCOs (dBm)	17
9	Max. output power, 2-4 GHz VCOs (dBm)	17
10	Max. volt., W175 into FM aux. (V)	2
11	dB/V slope, W175 into AM aux.	5.5
12	W$ data, min. voltage (V)	0.1
13	W$ data, ΔV between entries (V)	0.2
14	Max. volt, W175 into AM aux. (V)	5.0

* As of 10 Sep 1981.

V$[120] is used by the "load$" subroutine blocks when transferring data from tape to controller. It may be assigned by the user if desired; however, the user-assigned contents would be lost when "load$" is called.

U$[36] is used or read by several of the subroutine blocks. It is used to pass data from subroutine blocks to the W175 arbitrary waveform generators, to hold an operator's entry in "inRFid," and to hold the calibration identification retrieved by "initial." U$ must also be used for any operator inputs that are to use the "enter" subroutine block (see Appendix C). The user may make other assignments to U$ as desired; however, the user-assigned contents would be lost if "swp175," "AM175," "DC175," "inRFid," or "initial" are called.

X[14] holds various long-term constants and subroutine block parameters, such as loop times, W175 voltage factors, and the fraction of a noise spot generated from the fill oscillator (see Table E-4). It may be dimensioned larger than 14 elements, especially if the user wants to save new long-term constants, such as loop times for user-written run type subroutines (see Appendix D). The user may change the X[*] contents (either temporarily during a test or permanently by retaping) to match changed simulator conditions, such as a slower nextval subroutine (see Appendix D).

Of special interest is X[5], which contains a fractional number (0-1) indicating what proportion of a noise spot should be generated from the fill oscillator, with the remainder coming from the noise generator. As part of a program, the user can assign new fractions to X[5] and so vary the results when "fnoise" is called with otherwise identical parameters.

The simple variables U to Z are used for a number of purposes by the subroutine blocks. The simple variables do not need to be dimensioned unless the user plans to record the variables on tape. Simple variables within subroutine blocks are chiefly used as for/next loop indices and to pass values from one subroutine back to another, which called the first (e.g., from a nextval to its calling subroutine).

When only one value is returned by a subroutine, it could be rewritten as a subroutine function and so would not need to use a simple variable. The indirect return through a global variable from one subroutine to the other is more directly applicable when more than one value is returned, though it would also be possible to pass extra p-numbers and have the values returned through those. The one real advantage of using simple variables to hold return values comes in checking software, particularly after a crash, since simple variables, unlike p-numbers, are not lost when the controller is reset.

The simple variable Z is used to set up error codes ("err stp") in a manner independent of the p-numbers; thus, assigning an error code to Z will not lead to overwriting some p-number whose value would prove useful in debugging that error.

The user may use the simple variables U through Z if so desired, subject to the usual caution that the user-assigned value will be overwritten and lost if a called subroutine block uses that variable. The user must avoid stacking for/next loops in such a way that an inner loop and an outer loop have the same variable as the index. This includes cases in which the inner loop is part of a subroutine called by the outer loop. For reference, the subroutine block uses of the simple variables are listed:

U - internal return,
V - for/next index ("fval#"), internal return,
W - internal return (especially "fval#"),
X - for/next index,
Y - for/next, internal return, and
Z - "err stp" code, internal return.

The data values held in Z$, X$, Y$, and W$, are not constants but are subject to change, due in particular to each VCO's voltage-frequency relation's drift with time. The data should be recalibrated periodically. When to recalibrate is an empirical decision, with once a week as an estimate. A calibration program has been written that will set up the simulator for calibration and that will properly manipulate and store output measurements. An operator may choose to recalibrate the entire simulator or just part of it (e.g., just recalibrate the tune data for one VCO).

Recalibration introduces the need to keep track of the calibration data in use, so that an operator can confirm that the data are current. The calibration program prompts the operator to provide a calibration identification line, which would usually be the date of the calibration. This identifier is saved with the data and will be printed on the HP9825's internal printer whenever "initial" is called.

A transfer program has also been written to have the controller transfer the data on one tape (e.g., the calibration program tape) to other tapes. With a little practice it is possible to completely recalibrate both VCO's in an RF channel in about 45 min.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX F

ERROR HANDLING

This appendix summarizes how the simulator subroutine blocks handle errors. The user's program may choose to handle errors differently, so in addition to summarizing the subroutine block behavior, the appendix will give a number of suggestions on different ways of handling errors. The reader should also read Appendix C, which contains notes and suggestions on subroutine parameter checking.

The subroutine blocks carry out a number of checks on the parameter values passed, chiefly to ensure that the values are legal and within range. Each subroutine's checks are independent, and it is up to the user to check combinations of subroutine blocks. For example, in setting a noise spot, the subroutine block ("fnoise") will check that the spot parameters are legal. It would be the responsibility of the user to ensure that the spot remains legal if the spot's tune center is changed (e.g., a 300 MHz noise spot could be legally set at 3.2 GHz but would be clipped and hence illegal if the center were changed to 2 GHz).

When an illegal value or other error condition is found by a subroutine block, there is no efficient way of having the subroutine fix that value. The subroutine blocks, being independent, cannot identify how the user got the value passed and so cannot return there to prompt the operator for a new value. Default values set by the subroutine blocks to replace bad values are unacceptable because they would give the user and operator a false idea of what the simulator is doing; even when defaults are reported, there remains the objection that the simulator would not be doing what the test wants as indicated by the passed values.

When a subroutine block detects an error, it will set a coded error report number in Z and then branch to the "err stp" label. The "err stp," properly speaking, is not a true subroutine but becomes part of whatever subroutine entered it. When entered, "err stp" will remove the simulator's outputs by setting the RF channel function control cards to the same state they are in after "initial;" carriers, fill, and noise are turned off. The 50 ohm outputs of the W175's are also turned off. This forces the operator to correct the fault, preventing the fault from being overlooked and the simulator from being used in a frozen state (output fixed as it was when the error was found).

After removing the simulator's output, "err stp" will format and print the error code passed in Z. This three-part code indicates the subroutine that entered "err stp" and the reason; it may also indicate the VCO number that was passed, as a clue to where in a program the bad call was made. (The controller presently has no capability to save the current line number, aside from deliberately causing a controller error so as to set the erl label. See the remarks toward the end of Appendix C). Table F-1 lists the subroutine block error codes.

Table F-1 – Error codes.

1. fset
   - 0 – illegal VCO number
2. fval#
   - *0 – frequency less than minimum
   - *1 – frequency greater than maximum
   - *2 – D/A number out of bounds
3. pulse
   - 0 – illegal VCO number
   - *1 – illegal source number
4. biph
   - 0 – illegal VCO number
   - *1 – illegal VCO number
5. fnoise
   - 0 – illegal VCO number
   - *1 – spot BW about center frequency out of range
   - *2 – illegal video number/BW
   - *3 – band part frequency out of band
   - *4 – spot BW too large
6. auxmod
   - 0 – illegal VCO
   - *1 – illegal source number
7. ampset
   - 0 – illegal VCO number
   - *1 – dB out of range

8. AMown
   - 0 – illegal VCO number
   - *1 – rate out of range
   - *2 – "AMval" return out of range

9. ownswp
   - 0 – illegal VCO number
   - *1 – dwell per step too high
   - *2 – dwell per step too low

10. setVCO
   - 0 – illegal VCO number

11. stepmod
   - 0 – illegal VCO number
   - *1 – "stepval" return out of bounds
   - *2 – "stepwt" return out of bounds

12. special
   - 0 – illegal band number
   - **1 – illegal table length
   - 2 – rate out of range

13. loadY$, – X$, – W$
   - 0 – illegal VCO number, – Y$
   - + 1 – 2nd VCO in same RF, – Y$
   - 2 – illegal VCO number, – X$
   - + 3 – illegal video filter number, – X$
   - + 4 – 2nd VCO in same RF, – X$
   - 5 – illegal VCO number, – W$
   - + 6 – 2nd VCO in same RF, – W$

14. swp175
   - 0 – illegal VCO number
   - *1 – rate out of bounds
   - *2 – illegal function number
   - *3 – deviation about center frequency out of range
   - *4 – required voltage out of range

15. DC175
   - + 0 – illegal VCO number
   - 1 – 2nd VCO in same
   - 2 – illegal W175 ident. number
   - 3 – % duty cycle out of range
   - 4 – period out of bounds

16. T/P
   - 0 – rate out of bounds
   - † 1 – illegal VCO number
   - 2 – 2nd VCO in same RF

17. AMaux
   - 0 – illegal VCO number
   - *1 – illegal source number

18. AM175
   - 0 – illegal VCO number
   - *1 – rate out of bounds
   - *2 – illegal function number
   - *3 – max. dB out of bounds

Notes: * Third part of code will be VCO number.
† Third part of code will be parameter list position number.
** Third part of code will be band number.

After printing the error code, "err stp" goes into an endless message loop, flashing an operator notice and beeping. There is no explicit exit from this loop. The operator must stop the controller to get out of the message loop. This forces the operator to take an active step to recover from such a crash.

Once the message loop has been stopped, further actions for recovery are up to the operator. The local p-numbers can be read by the operator as an aide in debugging, if the controller has not yet been reset. The controller should be reset before continuing on to fix the test in order to clear the subroutine address return pointers.

Similar to "err stp" is "shutoff," which will perform similar simulator shutoff, fault reporting, and message loop functions when the controller detects an error. The user must enable "shutoff" as an error recovery routine if it is to be used; otherwise, controller errors will result in the controller stopping with the simulator output frozen at its state when the controller fault was detected. To enable "shutoff," the user should include this line in the program's initialization:

```
on err "shutoff"
```

Recovery from "err stp" or "shutoff" depends on the details of the user's program. The operator can always rerun the test from the beginning, making the necessary corrections in the rerun (if there are corrections to make; recovery from a controller fault due to tape read errors is usually a matter of trying again). If the operator knows of a program location from which corrections can be made and the test continued rather than rerun, this would save time and tape wear. The user may provide a convenient mnemonic label (i.e., "start" or "recovery") that the operator can use.

It should be noted that neither "err stp" nor "shutoff" affects the Z[*] contents, so the simulator could be restored to its precrash state by sending the Z[*] contents out to the channel function control cards and (if needed) tuning on the W175 50 ohm outputs. The Z[*] contents can be sent to the function control cards by using any subroutine that affects those cards, or the Z[2] through Z[12] contents could be sent directly to the multiprogrammer.

Error handling by the user's program can be far more extensive than simply providing a label for an operator to use. It is, of course, easier to catch errors before they result in a crash. The user can provide extensive checking of operator inputs to catch bad values before they are passed to the subroutine blocks (see Appendix C).

The user can also check combinations of parameters and prompt the operator to correct any faults or conflicts. Of particular interest is the allocation of the two W175's among the three uses of FM, AM, and pulse. The user should track such W175 use to make sure that no conflicts arise (e.g., if the AM W175 is used for AM, the user should make sure that an RI channel's pulse circuit is not actively connected to that W175 because the waveforms and voltages may not be compatible). Usually such device tracking is implicit in a program.

The user can also provide more extensive error recovery routines to replace "shutoff". The user might allow recovery from certain errors (e.g., can repeat a tape load a fixed number of tries if a tape read error is found), with unrecoverable errors handled by branching to "shutoff" much as the subroutine blocks branch to "err stp."

The amount of error checking provided by the user may range from none to very extensive. Users will probably provide checks and recoveries for any faults or errors considered likely and leave the rest to the subroutine blocks and the controller. There is a trade-off effort between recovering from an error without user-provided help and in the user's efforts and time in providing that help. Trade-off choices are up to each user in each test program.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX G

TAPE USE

Programs and data for the HP9825 controller are saved on a magnetic tape cartridge. This appendix describes several restrictions on and suggestions for tape use. The user is assumed to be familiar with the basic tape use instructions as given in the HP9825 reference manual.

Tape use is restricted in that files 1 through 91 on track 1 are reserved to hold the subroutine block data (see Table 6). The reservations require about 20% of the available tape on track 1. When a new tape cartridge is prepared for use, the data files must first be marked. A convenient, easy-to-type instruction that will mark each file is:

```
trk 1; rew; mrk 1, XXXX;
  mrk 6, 800; mrk 84, 150;
    mrk 1, 60.
```

This instruction can be typed and executed as one line from the keyboard. The string files will be a few bytes larger than is actually necessary; this is not significant. The file that holds X[*] should be marked somewhat larger than needed to allow for future growth such as the addition of user-written loop times (see Appendix D). File 0 of track 1 is available to the user; in the marking line above, XXXX stands for the user-selected size of this file.

When the subroutine block data files have been marked, the contents can be loaded using the transfer program and any older tape (calibration or test program) that has the data to be loaded. When the transfer is complete, the user may record the transfer program itself on the new tape for possible future use.

In addition to the data files on track 1, the user must mark a file somewhere to hold the subroutine blocks. The actual file locations and track are up to the user. Typically a new tape will include one master file containing all of the subroutine blocks, and it may contain other files holding smaller subsets of the subroutine blocks for use in programs where memory efficiency is important. File 0 of track 1 would be a logical place to put the master subroutine block file, if the file is marked large enough.

There is a use for file 0 of track 0 that, while not required, is suggested as a good way of identifying tapes and providing at the same time against a power on/off cycle leading to an undesired test. If a tape cartridge is in the controller when the power goes from down to up, the controller will automatically attempt to load and run the program contents of file 0 of track 0. A tape (especially one containing several tests) should generally avoid putting a test program in that file in order to avoid unintentionally running that program if the controller's power is cycled on/off/on with the operator busy elsewhere.

The autoload and run are useful, however, in cases where the user wants a certain program to run without that program having to be separately loaded. Demonstration tapes might use this feature to start their demonstration as soon as power is turned on. More generally, file 0 of track 0 can be used as an index to the rest of the tape. Such an index could identify the test programs available on that tape and could include some general information on their use. The index can be written to allow the operator to select which test to run by naming a menu selection, with the necessary load program (ldp) being carried out by the index program.

An index program can mix use of the controller's display and printer. A typical arrangement would use the printer to list the available test programs and their file numbers, and the display to give information about those tests. When using the display in an index program, the user can allow the operator to self-time the display by using the stop instruction rather than wait after each display instruction. The printer or first display can prompt the operator to press CONTINUE to self-time the following messages. This is similar to what the user can do in a dedicated help type information program (see Appendix C).

If stop is used, two points should be kept in mind. First, the operator should be given a definite indication when the end of the file 0 contents has been reached and should be told to keep pressing CONTINUE until done (so the operator does not assume the index is done before it actually is and so lose some information). Second, when mixing print and display statements, the user should arrange them so that the display is not wiped out when the printer is used. This can be done by simply not using stop after a print.

The user can also indicate the contents of a tape by using the HP9825's tlist instruction. The paper tape list of the magnetic tape files can then be marked by writing each file's name or purpose next to the file number. This paper would be kept with the magnetic tape and could be used by an operator as a quick guide to what file to load to get a desired test.

The basic idea behind using the tlist paper or file 0 of track 0 or both is to give the operator a ready indication as to the tape contents. The operator should not have to load a file in order to find out what it contains.

When the user marks a new file in order to record a test program, data file, subroutine block subset, or other program element, it is recommended that the file be marked larger than is actually necessary. This would allow for easy expansion if the program element to be taped is later modified. The amount of excess marked space should increase as the required size increases (e.g., a file for a 700 byte element might be marked as 1000 bytes, while a 9800 byte element could be marked for 12,500 bytes). The user should avoid having to mark a new file (or remark an entire tape) if the contents of a file are later modified to be larger than the marked size. It is quite easy to have a 9K byte program grow to 10.5K bytes, which would be a problem if the file for that program had been marked for 10K bytes. However, no file need be marked larger than the total available memory size of the controller (currently 22,910 bytes).

If the user writes a long program or needs a large data table for a run type subroutine (e.g., "special"), the program can be written as a number of separately taped segments, so that at any given time only the needed segment would be in controller memory. This differs from a program that may be taped in several segments but runs as one (e.g., separately taped program and subroutine blocks). A program written and taped so that only part of it is in memory at any time can be termed a multisegment program.

Multisegment programs would be ordered so that each segment contains a complete interval or several intervals. The user should be aware that loading a file can take some time, the actual time depending on the length of the file to be loaded and the time needed to find that file, which depends on where the track head happens to be when the load is called. Timing considerations may be used in deciding the intervals in a segment.

In using a multisegment program, the user would link the segments with the load file (ldf) instruction, which keeps data values intact by continuing after the load. The load program (ldp) instruction runs after the load and so would wipe out all data values and dimensions. This effect of ldp may be useful in passing from one independent program to another since it lets each program use whatever variables it needs, independent of the other program. An index program as mentioned above would be a good place to use ldp rather than ldf.

When using multisegment programs, the user must avoid overwriting any instructions that should be left in the controller. This can be done by using explicit line numbers in ldf for the load line and the continue line. This does require the user to keep explicit track of the right line numbers since the HP9825 controller presently has no readily accessible capability of tracking line numbers.

The explicit line numbers can complicate program modifications that add or delete (or otherwise change) line numbers. The user in such cases must avoid unintentional returns after a modification. One way would be to use a number of null lines. The load and continue line numbers could be located in the middle of eight or ten null lines, so that addition or deletion of a few lines will not hurt if the user forgets to change the line numbers.

Whenever the user has tape operations inside a program (data loads, program files, or multisegment changes), it would be a good idea to specify explicitly the track number rather than specifying it implicitly. This amounts to including the track statement with every tape operation, rather than only when it changes from its previous value. This is especially desirable whenever the operator can affect the tape track (which can be done from the live keyboard). Much of the time this would be an extraneous step, but it can avoid ambiguity and incorrect track use.

The user may also include steps to ensure that the tape track is known when the tape is removed from the controller by putting the tape to track 0 after the last tape use. Determining when and where the last tape use occurs will, of course, depend on the details of the user's program.

There are also a few suggestions the user can follow to increase tape life and reduce read and record errors. After the last tape use, the program could execute the rewind instruction. There should be no noticable time penalty to this, and it will cover against the operator failing to press the rewind key before removing a tape. Tapes should be rewound when removed from the controller for any length of time in order to protect the tape and its contents.

The user should avoid putting all files on the first few inches of tape or putting a heavily used file there because doing so would lead to increased tape slack and decreased life. The user may simply dimension file 0 on track 0 very large (say, 10K bytes), with the program files following on the same track.

Tapes should be completely rewound occasionally or whenever visual inspection of the tape cartridge shows the tape is wound unevenly. The HP9825 users guide has suggestions on how to do this. The user can also erase the tape from the last numbered file on either track and then rewind the tape. If this is done, the user (or operator reminded by the user) should be sure to specify the track number and null file number, or, obviously, unpleasant results could follow.

The user is urged to prepare backup copies of every program tape and to keep those copies separate. This provides protection against tape loss or destruction. It also allows the user to modify a file and try it out on a test run before copying it to the backup tapes, so that if the modification does not work or is undesirable, a copy of the unmodified program will be available.

If a tape does not have any files recorded by a program during a test (i.e., if nothing is written to the tape after the user prepared it), the tape can be protected from accidential use of record file (rcf) for load file (ldf) by using the slide tab on the tape cartridge. Using the tab means that the tape cartridge must be consciously prepared in order to write new file contents onto it.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX H

W175 ARBITRARY WAVEFORM GENERATOR USE

This appendix describes some details on the use of the W175 arbitrary waveform generator as part of the simulator. It does not serve as a general guide to the W175; for that, the user is referred to the W175 instruction manual (especially Sections 1 and 3). It will be assumed in this appendix that the user is familiar with the W175 and knows how to set it up. The user may be reminded that the letter character codes for the various W175 data are printed in the lower left corner of the corresponding key, which provides a convenient reference for those codes with typing instructions to the simulator controller.

W175 users may specify the state of the arbitrary generator either by means of a coded string sent over the bus or from the W175's front panel. The front panel is easy to use and lets an operator rapidly and randomly change the W175 settings. This is useful when the operator wants to observe the effects of different settings without having to go through the controller and is appropriate when the user is mainly concerned with checking the resulting waveform in order to observe its characteristics.

However, the controller cannot check any W175 settings made through the front panel. Some W175 settings that are legal as far as the arbitrary generator is concerned are illegal in certain simulator states. Examples of this include feeding a bi-sign (+/−) voltage waveform to a pulse circuit and setting too high a voltage for an auxiliary FM input.

Whenever possible, all W175 parameters should be checked and set through the controller. The additional programming workload that results will be offset by increased confidence that the W175 setting is legal and meaningful. Using the controller to set all W175 parameters also ensures repeatability and lets the controller track and document the settings.

On paper, the controller can prevent a bus device (the W175 front panel) from being changed by an operator by sending the local lockout statement (e.g., llo7) to any device that will respond to that bus line. It could later re-enable the device (front panel) by sending the clear lock/out local statement (e.g., lcl 701). Peculiarities have been noticed in practice with the W175 front panel. When a program is running, the W175 front panel can be freely used up to the time that some message (other than bus clear) is sent to it. After that, the addressed W175 is in a remote mode and the front panel keys cannot be used to change any settings. Lockout need not be sent. The front panel can be re-enabled by the lcl statement. This much is expected and follows the bus response description in the W175 instruction manual. A peculiarity is that, when lcl is used, it has been found that the W175 will not respond to any following bus messages, unless the controller is reset. This holds regardless of whether or not llo is sent after lcl.

To restate this, the controller will lock out the W175 front panel as soon as it sends any message specific to that W175 (clr 701 will not cause a lockout, but sending "ZI" will). If the lockout is ended by the device clear statement lcl (which the operator may send independently of a program by using the live keyboard), the W175 will then ignore any further bus messages.

In order to avoid lost W175 bus messages, the user's program should not send the clear lockout/local (lcl) statement, at least not if there are or could be later bus messages. If lcl is ever sent, then all further changes of that W175 must be made by the operator, until the controller is reset (which must reset the test program as well). Nor should the operator send lcl through the live keyboard, unless the operator is aware of the result. The user could disable the live keyboard to prevent the operator from sending lcl, but doing so would keep the operator from using the live keyboard for anything else (including test control, especially if the program uses the controller's function keys for anything), so disabling the keyboard is a discouraged option.

Any trouble with the lockout and local characteristics of the W175 can be avoided by having all W175 parameters sent through the controller; however, there may be cases in which the user wishes to let an operator vary some settings. The program could give the operator time to use the front panel by stopping between initialization and the first W175 bus messages, with the operator pressing CONTINUE when done. This approach does have the advantage of being easy to program and is quite short. It is basically a one-time item for each test run.

A better way of having an operator set up the W175 would be for the operator to specify the con-

tents of the message string sent over the bus. This can be done with the live keyboard and a quoted character string if the operator knows the right bus address (see Table H-1). It can also be done by having the controller prompt the operator. If the operator is knowledgeable and the user wants or needs a short, easy-to-program input, the operator could enter the message string directly, with addressing and perhaps some simple checking and manipulation done by the user's program. If U$ or V$ are used to pass the message (which would save the user from having to dimension a new string), they should first be cleared with null assignments (e.g., "" → V$). The user could make sure the string contains upper-case characters by using the cap function and could check that the last character is the W175 execute symbol ("!"), adding it if it were missing. The user's program might also prompt the operator through all the W175 options but this would require far too much programming effort and space to be practicable.

There are three uses for the arbitrary waveform generators (FM, AM, pulse) but only two W175's, so the user's program must allocate each W175 between FM and pulse or AM and pulse. If a test would require all three uses, the program must find a different source for one of the three. If an external source is not available, the simulator's multiprogrammer cards can be used. For example, FM and pulse signals could be taken from the W175's, with AM being run through the controller ("AMown") using the D/A cards in the HP6943 extender. If a pulse signal is a 50% duty cycle square wave, it can be generated through the timer/pacer card, and, if it is at a 10 or 100 Hz rate, it could be taken from the RF channel pulse sources, leaving the W175's free for FM and AM.

What the user wants to avoid is having a W175 set up for one use with the resulting signal being fed to the wrong circuit on some RF channel. This would give a confusing and incorrect simulator setup. The waveforms, rates, voltages, and offsets suitable for one modulation type use are unlikely to be suitable for the other type use.

If a W175 is set as a pulse source, its 50 ohm output would be turned off. If the pulse source W175 were used as an FM or AM source in connecting an RF channel through the auxiliary switch matrix ("auxmod" or "AMaux" with a source number of 3), there would be no actual FM or AM (aside from any leakage). If a W175 is set up for FM or AM, the 50 ohm output will be on, but there is no way of turning off the 0 ohm output. Thus, if the FM or AM source W175 were used as a pulse source in setting an RF channel's pulse circuit ("pulse" with a source number of 3 or 4), the FM or AM signal would be fed to the pulse circuit. This would not damage the hardware, but it would give an incorrect idea of the simulator status, especially to anyone monitoring the simulator status LED's but not the RF output.

It is important that the user keep track of the use of each W175 in order to avoid conflicts, especially when a W175's use may be changed during a test. Tracking is usually implicit in what the program does but may need to be done explicitly, especially when an operator controls the sequence of test intervals. Tracking can be done by setting or clearing the controller's flags (or any other user-assigned variables) to indicate the use.

When setting a W175's function block frequency or rate, the user should keep two things in mind. First, what is actually being set is the sample time per block point. If the block size were changed after setting a block rate, then the block rate would change, since the output would involve the same sample time per point over a different number of points. If the block rate and block size are changed in the same message string, the order is important. That the block rate will change when the block size does can be exploited as a quick way of changing the rate, since it takes about 25 ms to change block size as opposed to 50 ms to change block rate (see Section 1 of the W175 instruction manual).

Second, the user should keep in mind that the block rate or block frequency is not the same as the modulation frequency. The block rate actually describes the rate at which the W175 function block contents are sent out. A modulation frequency describes the rate at which the output modulation is changing.

The modulation rate of a W175 output is a function of the block rate, the output amplitude, and the

Table H-1 – Bus addresses.

Address	*Device*
723	HP6942/3 multiprogrammer and extender
701	FM W175 (B)
702	AM W175 (A)
703	492P spectrum analyzer (optional)

Note: HP6943 extender is at frame number 100.

function block contents (the output waveform). This last dependence makes it difficult to specify modulation rates directly. The function block contents are not generally known if a RAM block is used. The RAM contents may not be continuous or substantially monotonic, may not be full valued (may not cover the full amplitude range set on the W175, with maximum Y value under +127 or minimum over −127), and could result in a nonuniform output modulation rate.

In some cases the modulation rate can be easily related to the block rate. Pulse rates (inverse periods) can be directly set, as can rates given in hertz. Modulation rates given in units per second (e.g., MHz/s, dB/s) can be readily handled if the function block contents are well behaved (would give a uniform output rate) and are full valued, and if the modulation deviation is known. The user would then find the block rate necessary for a desired modulation rate as a relation of the number of times the output covers the modulation deviation range when the function block is sampled and of the amplitude of the deviation range. The relation is

$$\text{block rate (Hz)} = \frac{\text{modulation rate (unit/s)}}{[k]\,[\text{modulation deviation (unit)}]}$$

where k represents the number of deviation swings in one block sample (this can be determined by noting the number of times the Y value of the block contents changes by an absolute value of 255 points). This factor will change if a partial block is used.

The user may find other relations to use if the block contents are not full valued (e.g., the user can scale the deviation range in the relation expression above). If the block contents are such that the resulting modulation rate would be nonuniform, it becomes more difficult to define and find a relation between block rate and modulation rate. It may be worth repeating that, if a modulation rate can be expressed in hertz, that rate can be used in setting the block rate (with perhaps a k number factor included, especially if a RAM block with several cycles of a waveform is used).

The minimum and maximum block rates will depend on the block size, since the real minimum and maximum of interest are those of the sample time per point. These limits are 200 ns (500 ns for RAM) and 999.9 s per point. Minimum block rates are unlikely to be any problem because it is difficult to think of any ECM simulator modulation requiring more than 71 hours for one cycle. Maximum block rate for a full block is 19.5 kHz (7.8 kHz for RAM). If a higher block rate is needed, a partial block can be used. The user should remember, however, that a partial block has less resolution than a full one. Also, maximum block rates should be considered in terms of the response of the modulated device (such as the 500 kHz pulse circuit maximum).

The subroutine blocks that directly or indirectly pass block rate to a W175 check the maximum rate under the assumption that a full non-RAM block is used, so that the maximum accepted rate is 19.5 kHz. If the user wants to pass a higher rate for use with a partial block, the subroutine blocks must be either modified or not used, or the user may use the subroutine blocks to pass a scaled rate, up to the maximum 19.5 kHz rate, as if for a full block, then change to a partial block when the subroutine returns. The scaled rate would reflect the size of the partial block.

At present, the subroutine blocks will accept a RAM block rate between 7.8 and 19.5 kHz even though such rates are actually illegal and would cause a W175 error (such an error would give an error display on the W175 but would not crash the simulator). It is up to the user to ensure that RAM block rates are below 7.8 kHz. Such RAM checking can easily be added to the subroutine block software at a slight increase in memory size (see Appendix K).

The subroutine blocks will accept any legal function number to designate the waveform function block (see Table H-2). It is up to the user to ensure that RAM blocks are programmed and up to the operator to ensure that PROM blocks actually have PROMs. If a RAM waveform is used very often or must be used at a rate greater than 7.8 kHz, the RAM contents could be programmed into a PROM (assuming the user has a PROM programmer available) and the PROM then used in place of the RAM.

The user will find the RAM blocks very useful whenever a nonstandard waveform is needed. When high time resolution is needed, the user can stack the RAM's to get up to 1024 time points for use in describing the waveform. The RAM's must be programmed separately and the same block rate (sample time per point) used throughout.

Two examples can help illustrate the usefulness of the W175 RAM blocks. First, suppose a test design called for a signal to blink with some particular pulse pattern, in which the pulse duration may vary from pulse to pulse, to be followed by a look-through or off period, the pattern then continuing with some overall period. A W175 can be used as pulse source with a single RAM waveform to handle the complete pulse behavior. The RAM Y values would be 0 or

Table H-2 – W175 functions.

Function No.	*Function*
0	Sine
1	Triangle
2	Square
3	Ramp
4 - 7	Single PROMs
8 - 11	Single RAMs
14 - 17	Stacked PROMs (1, 2, 3, or 4 stacked)
18 - 21	Stacked RAMs (1, 2, 3, or 4 stacked)

Note: Stacked RAMs must be individually programmed.

+ 127 and the X positions would specify the actual overall pulse form; the block rate would be the inverse of the overall period. Amplitude and offset would be 2 and 1 volts, respectively, and the 50 ohm output would be off. The setup would be somewhat similar to getting a stable pulse waveform using "DC175" but more flexible. The user could thereby get the complete blinking and look-through/off behavior without involving the controller other than to set the W175. Of course, if a real look-through with measurements were carried out, the controller should handle the off timing in order to know when to make the measurements; the W175 example, though, would be an easy way to simulate such behavior.

Second, suppose a test design requires a very small noise spot (one smaller than the residual modulation when the noise generator attenuator is set as 7 or full attenuation). The fill oscillator output alone could be used, but not if good noise characteristics are needed. The user could get a small spot with noise characteristics by using the FM W175 as a noise source.

The user would first program the RAM blocks to contain random Y values in the range – 127 to + 127. This can be done easily with a simple for/next output loop. For best results, all four RAM's would be programmed, though fewer could be used. An example of setting up the four FM W175 RAM's for noise is:

```
for I = 8 to 11: wrt 701, "C", I, "I"
for J = 0 to 255
wrt 701, "X", J, "Y", – 127 + 254 rnd (1)
next J; next I.
```

The W175 noise spot can then be used by calling "swp175" with the desired spot bandwidth as the frequency deviation and with the maximum block rate (7.8 kHz for RAM), specifying the RAM block(s) in which the noise data were stored.

Noise data could also be specified to meet desired characteristics conditions, such as Gaussian noise, filtered noise, and so on. The user may wish to program PROM's to hold noise data; this would allow the noise to be set up with a 19.5 kHz block rate rather than a 7.8 kHz rate. The highest possible block rate is desirable in order to most closely simulate a noise source.

This brings up a closing point applicable when the W175 is used for FM. The voltage-frequency data in the W$ calibration tables (see Appendix E) are taken from operator measurements during a run of the calibration program. That program uses the W175 sine block at a 19.5 kHz rate. This gives an output shape that is easy to measure on a spectrum analyzer display.

It has been noticed that when the FM W175 data are used to set up an FM sweep at much lower block rates (such as 1 Hz), the sweep bandwidth will differ somewhat from what it would be at higher rates even though the voltage is the same. An empirical rule to quantize this effect has not yet been worked out. When a rule is found, it will be included in a modified version of "swp175." At present, it is up to the user to check on and allow for the bandwidth rate effect.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX I

USER DOCUMENTATION

The user should document any newly written test program. Documentation will allow others besides the original user to read, understand, and modify a program. It will also make it easier for the original user to modify the program, especially if some time elapses between writing and modifying. Good documentation makes it easier to use subroutines and procedures from one test program in other programs and so can save time in the long run. In addition, documentation provides a means of recovering a test program in the event that all taped copies of that program are lost, destroyed, or found unreadable.

Documentation might range from little more than a program listing with notes to a full formal report. How much effort the user puts into documenting any test report will depend on the user's perception of the test's importance, its usefulness as a source of material for other tests, and the identity of the test's potential users (additionally, in practice the amount of time the user has for documenting tests would be a major factor).

At the least, documentation should always include an outline of what the program does, identification of what the variables used stand for, and a program listing. This much could all fit on the same paper. A listing can be taken on the HP9825's internal printer and the printer's paper tape saved by fastening it to sheets of notepaper, with the program outline indicated through program mnemonic label names (see Appendix D) and variable use notes pencilled next to the listing.

This much documentation, which can be done in very little time, is suitable for the personal use of the original user who wrote the program. It is not really suitable for use by others but would be quite acceptable for a simple program operated by the original user. Its advantage is that it can be quickly prepared from material the user would prepare anyway and so does not require any noticable additional effort or time from the user.

It may be appropriate to point out here that the HP9825's internal printer is a thermal dot matrix printer, using treated paper. The printing will fade with time, especially if left exposed to strong light and heat (if the printer paper is attached to notepaper with cellophane tape, any printing under the tape will fade much more rapidly). With any program listing kept more than a few weeks, it would be a prudent measure to copy the listing to a less perishable medium. Photocopying works well.

A listing with a note-type of documentation is what the user would prepare for personal reference. A more detailed documentation effort is needed when a test program will be read and modified by others than the original user. Better documentation is also desirable when a program is very long or very complex, or when it might serve as a source of subroutines and procedures for use in other programs. More extensive documentation than the basic listing with notes would be valuable if a program is modified after it was written, even if that program is operated and modified by the original user.

Such documentation can range from what would be a listing with expanded notes to a formal report with an operator's guide. As a rule, informal documentation should be quite acceptable if the program will be run by the original user, or by a few operators who can be shown through the program by the original user and who can directly ask that user any questions that arise while running the test program.

A formal report would be called for if a test program will be run by several operators, or when the program will be used outside the facility where it was prepared, or when a test would be used for some time (this last is a key point since the original user may not always be readily available to explain a program).

In such cases, it would be more efficient for the original user to prepare formal documentation once, rather than explaining a program to each new operator or to each new user wanting to modify the program. Formal documentation implies a more detailed account of what a progam does, such as giving a description of any algorithms or calculations in the user's program. Formal documentation would also include an operator's guide detailing the responses and options available to an operator.

The form of user documentation may vary according to what each user feels best meets the needs for a program. It is possible, however, to suggest a standard form for formal documentation. A standard form would have advantages in that it would give the documentation reader an idea of what the

reader can expect to find in the documentation. It would also save the user a little time and effort since the user would not have to devise a new form for each documentation report.

Informal documentation can use elements of the same standard form, dropping some sections, abridging and combining others. By doing so, the user makes it easier to upgrade from informal to formal documentation if the need arises.

The suggested standard form for documentation is made up of 11 sections, which are described in the remainder of this appendix. The standard form outline is:

1. Statement of test purpose,
2. Interval form/time behavior,
3. Operator inputs,
4. Fault recovery,
5. Tape use,
6. User subroutines,
7. Operator's guide,
8. Variable use,
9. Miscellaneous information,
10. Detailed description, and
11. Listings.

The statement of test purpose can be a single paragraph identifying the program's name and its purpose. The program name should be given as a label on line 0 to help identify the program when it is loaded in the controller or listed.

The interval form/time behavior identifies the test output as a function of time (see Appendix B). This will inform the reader what the test program does and identify output changes and what controls the timing and nature of the changes (such as a program-fixed change or one selected by the operator).

Operator inputs would be a list of what the operator can affect in the program. It is not an operator's guide (see below) but more of an information source, letting the reader know what an operator can enter and something of when and how entries are made, but leaving details to the user's guide. This would be on the level of "the operator may specify frequency centers and noise spots before the output begins by replying to controller prompts."

Fault recovery would explicitly tell an operator how to recover from a simulator crash, whether due to the subroutine blocks ("err stp") or the controller ("shutoff"). The user might provide a suitable label at some part of the program and inform the operator that the program could be continued at that label if a fault stops the program (e.g., the operator might type and execute something like 'cont "restart" '; see Appendix F). This part of formal documentation may also summarize how to fix any faults that might be expected in the test involved, such as specifying too large a deviation range at some point.

Tape use should identify the tape files needed by the program being documented. There will be at least the program file itself; there may also be data files, function key files, seperately taped interval segments, and help files. All such user files should be identified, giving track and file numbers. If the subroutine blocks (or a subset of the subroutine blocks) are loaded into the controller during the program, the track and file numbers for the blocks should also be identified. This part of a formal documentation report lets users and operators know what tape files are needed and where those files should be on tape. A list of tape files and purposes would usually be acceptable, making this a formal version of the information that the user might note on the paper returned by the tlist instruction and kept with the tape cartridge (see Appendix G). The operator can also be told when to remove a tape cartridge and reminded to rewind tapes before removing them.

User subroutines can be listed by giving their label names and parameter lists, with a brief account of their purpose (as is done for the subroutine blocks in Appendix N). This would provide a handy reference if a latter user wanted to see if any of the program's subroutines could be lifted for use in a different program. If good mnemonic label names were given to the user's subroutines, this part of a formal documentation could also indirectly give a short outline of the program structure.

An operator's guide would take an unfamiliar operator through all the steps required to get the test program running. It treats all parts of a program visible to and requiring responses from the operator. It can give more detailed information than will fit into operator prompts as part of a program (see Appendix D). This part of a formal documentation would begin by specifying what tape file to load in order to start the test program; it would then identify each operator input, giving the input's purpose, range, and default value, if any (see Appendix C). The reader would be told what conditional next steps would follow each branch control question. A reader should be able to go through the operator's guide of a formal documentation report and find out before running the test what the response would be to any input entry. A good account of the default values (set

by pressing CONTINUE after a prompt without entering anything) would be particularly useful. The user may find the operator response format used in the HP9825 Utilities Program reference book to be a useful model.

Variable use would identify the purpose the user assigns to the global variables (p-numbers within a user-written subroutine can be described as part of a detailed description; see below). The user should identify what each variable is used for, how the variables are set up, and what changes take place and when. If arrays or strings are used to hold data, the user should identify the overall purpose of the array or string and its internal structure (e.g., if A[*] is used to hold parameter values, the user states so and also lists the purpose of each element in A[*]). If any data are prepared and taped in advance of a test, the user should state how to prepare those data and where to tape them.

Miscellaneous information is a catch-all for any points the user thinks could be useful for the operator to know. Examples would be the use of the function keys in a test, notes on timing or accuracy constraints, possible program modifications, suggested test values, and output monitoring suggestions.

A detailed description of the user's program would be used chiefly by anyone who wants to modify the program. The detailed description would go through the user's program and account for each instruction and statement. It would describe the program's structure, its algorithms, and branching decisions; detail the timing control; and give the purpose of user provided subroutine p-numbers. This detailed description can be organized to match a program's structure. If a program is structured as a number of subroutine calls, the detailed description can first describe how the calls are made and how the passed parameters are found, and then separately describe the user provided subroutines. A detailed description would be the most time-consuming part of preparing a formal documentation report.

A copy of the user program listing would be included in a formal documentation, possibly as an appendix. This listing provides insurance against all tape copies of a program being lost. In case of such a loss, the program could be retyped from the listing (inasmuch as such typing can be remarkably tedious, the user is again urged (as in Appendix G) to make backup tape copies of all programs used more than once). The user will probably prepare a listing with a note-type of documentation as an interim measure when writing and debugging a long or complex test program; this can be directly copied (after any corrections) as the program listing, which would make the notes readily accessible to anyone reading the listing.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX J

INITIALIZATION STATUS

After the simulator power is turned on and the multiprogrammer completes a self-test, the simulator will be in the state shown in Table J-1. In particular, in each RF channel the pulse modulator is turned on, the biphase circuit adds a 20 MHz comb, the fill oscillator is off, there is no output power level attenuation, and the VCO select circuit may be ambiguously connected. Also, the states of the FM and AM auxiliary switch matrices are not definitely set but may wake up randomly, and the data format of various multiprogrammer output cards is not that required by the subroutine blocks. The result is that the simulator is in an ambigous state and its output consists of spurious signals. This result will also hold whenever the "clear bus" message is sent from the controller, except that the auxiliary switch matrices are not changed from their previous states.

The "initial" subroutine will remove the spurious outputs and ambiguous conditions and put the multiprogrammer cards into the data format needed by the subroutine blocks. After "initial" is called, there will be no simulator outputs. All modulators are off, attenuators are set to full attenuation (this includes output power level), and all switches are turned off. The subroutine will also intitialize Z[*] and print the current calibration data identification on the HP9825's internal printer. Before calling "initial", Z[*] and U$ must be dimensioned.

Once "initial" is called, the user may proceed to build up a test modulation from a known starting point (see Table J-1) and can rely on the multiprogrammer cards accepting the subroutine block data format.

Table J-1 – Initialization.

	Power Up	*Bus Clear*	*"Initial"*
Multiprogrammer D/A card formats	Unsigned binary		2's complement (slots 2-12)
Card data	0		32760** (slots 2-12) 0 (other slots)
Tune frequency end			
Low bands	Low frequency		
High bands	High frequency		
Noise generator	On/no attenuation		Off
Noise video	0		
Fill oscillator	On/no attenuation		Off
Pulse control	Carrier on		Off
Biphase control	20 MHz comb		Off
VCO select	B*	B*	B
Auxiliary switch matrices	Random	Unaffected	Off
Power level set	0 dB attenuation		8 dB attenuation
Timer/pacer card mode	1-shot		Recirculating
W175 RAM contents	0	Unaffected	Unaffected
6942 clock	Reset		

" " Indicates same as power up condition
* May be ambiguous
** Decimal equivalent of the octal pattern 0, 7, 7, 7, 7, 0

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX K

SUBROUTINE BLOCK MODIFICATIONS

There are a number of recommended modifications that could be made to the subroutine blocks as they exist as of October 1981. These modifications fall into three categories. First, some modifications could be made to improve the usefulness and structure of the subroutine blocks. These would serve to improve the existing subroutine block software. Second, some modifications could be made by a user for a particular test; such modifications are test specific. Third, some could be made to extend the subroutine blocks' capabilities, especially by modifying a one-VCO run type subroutine to cover several VCOs. These are extension modifications. For clarity, each of these three modification types is discussed under a separate subheading.

Improvement

Some modifications could be made to make the subroutine blocks easier to use and understand. None of these modifications is actually required, but some will be made in the future to obtain their particular benefits.

When checking frequency limits, the subroutines might obtain the numbers from Z$, using VCO number as an index and the num function to obtain the actual numbers (which must be scaled to hertz units). This would be longer and slower than the present use of limits calculated from the band number but would let the limits represent the actual achievable hardware limits (though this would complicate the calibration program since its operator would then have to indicate when the output does become clipped). It would also free the limit checks from a dependence on having a particular type of RF channel in a particular slot. It should be noted, however, that while on paper this would better match the statement that the MMG ECM simulator channels are interchangeable, it is not clear that in practice that interchangableness extends to removing one channel out of the rack and replacing it with a different channel type for one test. Removing an RF channel requires removing a number of screws and connections; it is easier to leave the channels where they are. When the type III RF channels are added, such a change may be needed.

A few changes are recommended for the "fnoise" subroutine blocks. Given its length, it may be desirable to divide it into two or three subroutines in order to make it easier to read. For example, the instructions that find attenuator settings from the fill and noise spots could be separated.

Aside from breaking up its length, a user may wish to modify the details of how "fnoise" actually gets an attenuator value. Presently the subroutine looks through the data for attenuator settings from 1 to 7 and tries to find the first data entry less than the desired output. If found, "fnoise" will use the previous attenuator value (0-6); if not, it will use 7. As familiarity with the simulator is acquired, it can be determined whether this method gives acceptable results; if not, the process can be modified. For example, the subroutine might look through 0 to 6, use the present attenuator value rather than the previous one (especially if it looks through 0-6 rather than 1-7), check for " less than or equal to" rather than "less than," or report a fault if the desired spot is less than the full attenuation value.

Two modifications could be made to "swp175," both of which would be useful. First, the instructions that calculated the W175 voltage based on the VCO and the desired frequency deviation could be split off into a separate subroutine (which might be called "voltswp" or "volt175"). This separate subroutine could then be called by the user's program whenever the user's program wanted to check if a W175 frequency deviation was legal (at present this must be done by directly manipulating data entries, which is awkward and requires fair knowledge of the data setup). This would be particularly useful when checking operator inputs.

Second, the check of the W175 block rate high limit could be conditionally tied to the function number to distinguish between RAM and ROM or PROM blocks (see Appendix H). This type of modification could also be made to "AM175."

If a simple empirical relation can be found between the frequency deviation at high rates and at low rates (see Appendix H), that relation could be worked into "swp175." Whether this is done depends on what magnitude the rate effect is found to have in practice and what sort of empirical relation is noticed.

It may be found desirable in practice to extend "loadX$" to allow video filter values (as well as numbers) to be passed. It may also be desirable to prepare a single subroutine that, when called, would call all three "load-$" subroutines using the same VCO number.

Test Specific

The user will occasionally find it useful to make some subroutine block modifications as part of a particular test. Such modifications would be used only in the test in which they are needed. It is important that the user keep track of such modifications, especially when the modified subroutine blocks are separately taped. It will be up to the user to avoid confusing the modified blocks with the original ones. Good tape documentation will help.

Probably the most likely test specific modification would involve "enter." If a test allows operator entries and does not expect any operator entries having "milli-" units (i.e., no milliseconds likely), then it may be desirable to modify "enter" so that "mega" units (i.e., MHz) can be entered with either an upper- or lower-case character, as is the case with "giga-" or "kilo-" units. This can be done by deleting the small m line in "enter" and adding the cap function to the large M line so that M is treated similarly to G or K.

If this change is made, it will save the operator some effort and make it easier to enter values without making mistakes. The operator should be told that a small m will specify "mega-", either in the documentation or as part of the program's initialization.

If it is found that this change is widely used in practice, it might be made a permanent modification similar to the improvement modifications described earlier. It might be done as a permanent modification anyway, so that an operator would not have to remember if a large or small letter M will do in a particular program.

Other changes are less likely to be considered permanent. If the user wants flag 14 set throughout the user's program, the clear flag 14 instructions in various subroutine blocks could be deleted (see Appendix E).

If space is short, the subroutine blocks could be abridged somewhat by cutting some of the checks (such as the VCO number checks). This should be done with care since any erroneous parameters (which are more likely in a long, complex program) could then lead to undetected error conditions. It would generally be better to use a subset of the subroutine blocks (loading only those needed) or a multi-interval with separately taped segments.

The user may on occasion also wish to change the error branching to allow use of defaults without crashing the program. This is acceptable if it is done as a test-specific modification, with any such defaults being reported (for example, by printing a notice on the HP9825's interval printer).

Some of the "?" calls are not always necessary and could theoretically be cut to save space, but it is generally safer to leave this call intact.

Modifications to the run type subroutine blocks can be carried out through the nextval subroutines if only the output values are involved. More extensive modifications are covered below.

Extension

Some modifications can serve to extend the capabilities of the subroutine blocks. These extensions may be modifications within the framework of the existing subroutine blocks, or they may involve such modifications as to create new ones. The latter particularly applies to creating new run type subroutines by building on and modifying an existing one.

The first group of extensions would involve a number of desirable features that might wait until the controller is upgraded and more memory space becomes available. Notably, the subroutines can be set up to keep track of the data in the controller and update then as necessary. A number of variables could be dedicated for use as flags, perhaps by extending Z[*] or using a newly dimensioned Y[*]; initial values would be set by "initial."

The extended software would primarily involve "setVCO" and "fnoise" tracking the VCO numbers and video filters in use. The "load-$" subroutines would be called as needed when either the active VCO in an RF channel or the video filter is used for noise changes. However, there may be cases in which the user might not want to spend the time needed to change the data (see near the end of this appendix). In such cases, the user would have to bypass or disable any automatic data updating.

The other subroutines might be extended so that when they are called they must be called with an active VCO number. Subroutine blocks that affect the tune centers or noise spot bandwidths can be

given common global variable references that can be checked to prevent illegal combinations, such as a 300 MHz noise spot, legally set on a 3.2 GHz center, being moved to a new 2.0 GHz center. Checks could be built in to prevent a W175 set as an FM source from being connected to a pulse circuit.

While such extensions would remove some of the subroutine block independence, the extensions would also increase the confidence that the simulator will always be in a legal state. Such extensions would slow down the subroutine block execution time and increase the required memory size by whatever is needed for the additional instructions and the data used to hold the checked values. It remains to be determined if such extensions are really needed or useful; such determination can be put off until such time as the controller is upgraded, giving more time in which to gain familiarity with the simulator and a better idea of what extensions would really be useful.

The second group of extensions involves modifying run type subroutines to get new subroutines. This sort of extension can be carried out whenever needed. With time, a library of run type subroutines can be built up, and the user would select only the ones needed for a test.

It should again be noted that if a run type subroutine needs only a different set of output values, the subroutine can be modified, assuming it uses a nextval approach, by simply changing the nextval (see Appendix D). If the subroutine does not use a nextval approach (i.e., "ownswp"), it would be rewritten to use a different means of getting its values (e.g., in "ownswp" one could change the triangular waveform to a sawtooth ramp by simply deleting one of the two inner X loops).

The most obvious case in which the user may wish to modify and extend a subroutine block would arise when the user wants a one-VCO run pattern to run on several VCOs simultaneously. The user can extend the single VCO subroutine. How this is done will depend on whether the VCOs, running simultaneously, share control cards and whether the individual VCOs, running simultaneously, change synchronously or asynchronously.

Generally, if the modulations of several VCOs share control cards, the changes must be connected through holding words; if there are no common cards, the changes can be taken independently. For example, if tune centers are changed, the user must remember that two VCOs are controlled by one tune card. Here the user must avoid accidentally wiping out one tune word of a pair when changing the other (unless the RF channel controlled by that tune word is not in use). This can be done as in "stepmod." Instead of sending out the new value directly, it is manipulated by binary operations to fit in an internal word, and that internal word (which also contains the data for the rest of the card) is sent out. On the other hand, the AM D/A cards are independent, so values for those cards can be handled separately.

In extending a run type subroutine block (or writing a new one), the user must determine if the individual sources change synchronously or asynchronously, and if asynchronously, if the rates are the same or different. All but the last can be handled in a fairly straightforward way.

If the sources in a multiple source run type subroutine change synchronously, all of the new data is sent over the bus at the same time, using the "output parallel" (OP) instruction. The desired output rate can be set with a single controller wait instruction.

It should be noted that the user cannot specify the same card slot address twice in the same OP instruction. If two of the synchronous changes are controlled by the same card, they must be combined before (or as part of) the OP instruction.

If the sources in a multiple source run change asynchronously but all at the same rate, the new data can be sent out using a series of OS instructions and waits. Each wait (offset by a loop time) is set to reflect the relative timing between each asynchronous change, subject to the constraint that the sum of all the individual waits (offset by all of the loop times) must match the overall dwell implied by the asynchronous rate. Since the rates are all the same, the relative timing between sources will be fixed. The user must specify the order and relative timing of the source changes.

It should be noted that, for the asynchronous changes, the loop time that should offset the wait between one adjacent pair of changes will not necessarily be the same as that for other pairs. Whether it is or not will depend on the details of the output loop. If a loop is set up to get all the values for one pass of the loop and then sequentially sends them out, the loop time between the last output of one pass and the first of the next pass would be somewhat greater than the loop time for pairs within the same pass. An asynchronous/single rate loop could also be set up so that it sequentially gets a value, sends it over the bus, and waits; in this case the loop times between any adjacent pair should be the same (assuming the same amount of time spent getting each value), but the loop time may increase.

Which approach is used for an asynchronous/single rate loop is up to the user. It might be slightly easier to keep track of tabled values if all values for one pass are collected at the same time. Of the two approaches, the former is better suited for rippling all the source changes close together in time, with a longer time to the next set of changes, since it would give the shortest loop time between changes within one pass. Unless it takes very long to get all change values, the former approach should also be satisfactory for patterns in which the changes are more symmetrically timed (i.e., roughly equal times between any two changes, including the last of one pass and the first of a following pass); if it does take too much time to get all the values, the user could first consider a different way of getting those values, such as calculation in advance of the output run, with the output values being held in an array.

On the other hand, it would be somewhat easier to determine the loop time offset when each asynchronous value is found separately, since then only one loop time need be found rather than two or more. However, getting all the loop values for one pass at the same time will save overhead time by reducing the number of times the loop must get a value. Also, it is not difficult to roughly estimate how much time each step in the loop should take. These estimates would then be refined by measuring the actual output dwells.

Some early simulator observations give rough estimates for the maximum synchronous and asynchronous output rates. Based on changing three RF channels and using data calculated and tabled in advance of the output loop, the maximum synchronous output rate is about 26.3 Hz. The maximum asynchronous/single rate output rate is around 12.5 Hz, but this could possibly be improved to about 16.5 Hz, since the 12.5 Hz estimate was made with a simple loop in which shift operations were carried out during the loop rather than in advance of the run.

It would be more difficult to run independent modulations at the same time, which is what asynchronous changes at different rates would be. The order in which the sources involved change will vary and the time between changes will vary. Some of the different rates may be multiples of some common base, leading to an occasional coincidence of changes for several of the sources.

In order to simulate asynchronous/different rates output, the user must be able to specify the order of source changes and the relative time between changes as functions of time. This will in general be difficult, but if it can be done the controller can run the asynchronous/different rates output.

Whenever possible, the user should try to tie several of the rates to a common base; the source changes run at those rates could then be defined in terms of their common output period. In other words, if some overall repeat pattern can be found, then the output can be organized in terms of that pattern. This is easiest if all the sources running are involved in the repeat pattern. Finding a repeat pattern would be the most straightforward approach to simulating an asynchronous/different rates output, but finding a repeat pattern (if one exists) can be tedious and difficult; if the repeat period is very long, an explicit use of that period may require too much space and effort to be practical.

Most generally, the user could try to devise some algorithm to describe at each output step which source changes, what value it changes to, and how much time elapses to the next change.

The same sort of subroutine could handle the actual running regardless of how the user actually defines the outputs. Basically the subroutine would determine how many output changes to run; then for each output it must determine an address (which specifies the source that changes), a value, and a wait. These values could be found using a nextval subroutine, where the nextval contains the user's definition of the output pattern. Any necessary checks or data manipulations (such as binary shifts and word combinations) should be done within the nextval and can be tied to the nextval's determination of the source or address. The basic form of the main subroutine can be outlined as:

```
determine N (number of changes)
for index = 1 to N
call nextval: determine
              address, value, time
write multiprogrammer: address, value
wait (time - offset)
next index.
```

This can, of course, be modified as the user sees fit. It would, for instance, need modification if any sources change simultaneously (perhaps by using a conditional jump). Also, each pass of the output loop might determine several address, value, and time sets and output them, with the loop index being changed to reflect the number of sets in each pass.

The maximum asynchronous/different rates output rate will depend on the number of sources and how complex the sequence determination proves, but

it might be expected to be fairly low in practice, with any individual source likely being limited to around 5 Hz when all sources have comparable rates. At the time this is written, no measured asynchronous/different rates measurements are available to check the estimated rate limit.

Summarizing what has been said so far about the extension of the run type subroutines: When extending a single VCO run type subroutine to handle several VCOs, the user must determine if the run type modulation sources are changed synchronously or asychronously; if the latter, the user must determine if the outputs are all changed at the same rate or with different rates. With asynchronous changes, the user must specify the order in which the sources change and the relative timing between changes.

Other extensions could be made for single VCO subroutines (these, of course, might then be extended to several VCOs). Two examples can be given readily. First, the user can write a subroutine to generate an FM signal through the FM D/A in the multiprogrammer extender by exploiting the obvious similarities with the AM D/A's. This might be called "FMown" and would be an analog of "AMown," just as "AMaux" is an analog of "auxmod."

Second, suppose a test called for sweeping a noise spot from a type II RF channel between 1 and 3 GHz (this assumes the 1-2 GHz VCOs would by then be available). The controller could run such a pattern in a manner analogous to that used in "ownswp." The sweep would start with one VCO active and the controller would move the tune center along its sweep pattern (whatever that happened to be) until it reached the 2 GHz value that separates the two VCO bands. The controller would then set the tune center of the inactive VCO to the limit value of 2 GHz and then switch the active VCO in the RF channel. The sweep would then continue. The necessary D/A numbers should be found in advance so that the data need not be updated during the run of the sweep. (The noise spot could be expected to change noticeably during such a sweep across VCO unless it were reset at a few representative points along the sweep; this would require the data to be updated, slowing down the sweep).

The basic point of this part of the appendix is that when a new run type output is wanted the user does not have to start from scratch but can save much time and effort by building upon the existing subroutines, modifying them to suit the new requirements. The actual nature of the modifications will depend on the needs of the user.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX L

PROGRAM CHECKLIST

The actual instructions that make up a test program will vary widely, depending on the nature of the test and the way a user chooses to write that test. However, some basic steps are common to every test program, as described in Appendix B. This appendix gives a brief summary and checklist of a few of those steps, specifically those needed to get any output from the simulator.

The simulator must be initialized: strings and arrays must be dimensioned, subroutine blocks loaded (if not already part of the program file), "initial" called, data loaded, and so on (see Appendix B). When the program is ready to create an output signal, it must ensure that the right VCO in an RF channel is active and call "setVCO" if it is not active ("initial" will set the B labeled, or higher frequency, VCO in each RF channel as the active VCOs). The program must then ensure that the biphase and pulse carriers are set. If no biphase (or pulse) modulation has been set for an RF channel, the carrier should be turned on. The tune center might be adjusted so that the output will be in the response range of that VCO (the wake-up D/A number of zero will in some cases give a severely clipped output that is out of the VCOs range). Finally, the output power level should be adjusted from its post-"initial" value of 81 dB of attenuation.

There is no fixed order for carrying out these essential steps (after initialization, which is the first step), but they must be done directly or indirectly if any output signal is actually to be found at the simulator's output ports. Some steps can be done indirectly; for example, setting a 20 MHz biphase comb will turn on the biphase carrier, and setting a noise spot through "fnoise" with a center frequency as the fourth passed parameter will set the tune center.

The following short list can be used as a programming checklist to ensure that the essential steps have been carried out.

1. Initialization
2. Set active VCO
   a. Implicit after "initial" if VCO number is even
   b. Explicit ("setVCO") if VCO number is odd. Update data.
3. Turn carriers on
   a. Pulse
   b. Biphase
4. Set tune center D/A number
5. Output power level ("ampset")

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX M

MISCELLANY

This appendix contains a miscellany of notes and suggestions that may be useful but that are not in themselves important enough or long enough to justify separate appendixes.

Memory Size

The 15 October 1981 version of the subroutine blocks requires 9970 bytes of the HP9825 controller's 22,910 available bytes, or roughly 44% of the available memory. Unneeded subroutines can be cut in a particular program to reduce the amount of required space: the size of each subroutine is given in Appendix K (the total of individual sizes does not match the combined size due to storage overhead). The variables and data used by the subroutine blocks require about 3.5K (15%) of the controller memory. The subroutine blocks could use about 13.5K (59%) of the controller memory if the full subroutine block set is used. This leaves about 9.4K bytes for the user's program and data.

Subroutine Block Placement

The subroutine blocks may be loaded in any part of the controller memory; there is no dependence on specific line numbers. Typically the user will write a program and add the subroutine blocks to the end of the test program. If a separately taped segment approach is used, it may be easier to load the subroutine blocks first and add the program segments below the blocks. If this is done, line 0 should have an unconditional go to branch to a label identifying the start of the user's program; otherwise, pressing RUN (or CONTINUE after a reset or editing) will lead to an execution error as the controller tries to run the subroutine blocks as if they were mainline programs.

f-Keys

The HP9825 has a number of user-definable special function keys or f-keys (keys for short). The keys work as part of the live keyboard, do not give an interrupt capability, cannot be used to call a function or subroutine, or use the ent statement in live keyboard.

The keys may be used during and as part of a test program. Uses might include reading the multiprogrammer clock and printing the values of some specified variables. The keys may run a for/next loop but should never use as the loop index any variable also used in the test program.

The keys can also be used to change the value of a variable by assignment of a new value. This feature might be used as a way of letting an operator rapidly change the rate of a run type output during a test; the output loop wait time would be assigned to a simple variable, and a number of possible values for that variable could be held in the keys; two keys might be set up to increment and decrement that variable value by some fixed amount each time those two keys are used. The feature can also be used to allow an operator to control branching: The program would set a variable to zero, then on a separate line use a prompt message followed by a jump of that variable, the keys assigning non-zero values to the variable. For example, the program might be:

```
0→A
dsp "select key"; jmp A
gto "first"
gto "second,"
```

while the keys would be:

```
f0: * 1→A        f1: * 2→A
```

The user will also find the keys helpful when typing a long program into the controller. Frequently used statements or combinations of characters can be stored in the keys (without the immediate execute asterisk) and the keys used to add the stored phases to the line being typed. The key contents do not have to be complete statements or meaningful in themselves. For example, when typing in a program that handles a number of operator entries, two keys would be:

```
f0: "" →U$; ent"
f1: ", U$[1, 32]; if flg 13.
```

Using the keys as a typing aid can greatly reduce the workload of typing a long program.

Fill On Noise Spots

This has been covered before (see Appendix E) but can bear separate emphasis. The "fnoise" subroutine block will divide an RF channel's desired noise spot RF bandwidth into a part taken from the RF channel's fill oscillator and a part taken from the noise generator. The fill is used to square off the spot's edges. Nominally, 20% of the desired spot is generated by the fill oscillator, with the remainder coming from the noise generator.

The fill can be used alone (noise generator off) by calling "fnoise" with an explicit video number of zero. The user (or operator) can change the fraction of a spot that "fnoise" will get from the fill oscillator by changing the contents of X[5] from its normal value of 0.2 to some other fraction before "fnoise" is called. The fraction should be in the range 0 to 1. A 0 will set the fill oscillator to full attenuation, leaving just the noise generator as the spot source. A 1 is not equivalent to passing a video number of zero, since the 1 would set the noise generator to full attenuation but leave it on.

How the value of X[5] (and hence the fraction of a spot generated by the fill oscillator) is varied is up to the user or operator. X[5] can be changed from the live keyboard; "fnoise" must be called before a change in X[5] will take effect.

Interrupts

The user is reminded that the bus interrupt handling capabilities of the HP9825 provide a powerful way for the controller to monitor the actual status of the simulator. At present this capability is not used and no simulator devices will actually interrupt the controller, but the user should not forget about this capability. Interrupts could be particularly useful if the simulator is modified and extended to include measurement devices to allow a closed loop monitoring of the simulator by the controller. The user should consult the HP9825 reference manuals (especially the manual for the extended I/O ROM) for details on handling interrupts.

Bus Device Status

This item is related to the above. The user should not forget the possible uses of the bus device status line. The bus status is explicitly read by the controller (e.g., rds 7, rds 723). It would be particularly useful in cases where the program sets up some bus device to do some slow or variable duration task and the program should do one set of procedures while waiting for the task to finish and another set after the finish. If the device sets the SRQ line, the status can be monitored to see when it is set. The HP9825 and 6942 (multiprogrammer) reference manuals can be consulted for suggested uses and examples.

Timeouts

The user may find the "time" function of the HP9825 controller useful if the simulator is extended by the addition of measurement devices. The "time" function allows a specified time for an I/O device to complete before "time" will cause an error. This could be used to check for damaged devices (those that fail to respond). It can also be used in conjunction with the on error branching of the controller to provide some alternate action if no I/O reply is made within a specified time. Several "time" periods could be looped to get longer periods by using the on error branch and a counter. The on error branch would replace or supplement the one that enables "shutoff". At present only the controller's keyboard could be considered an I/O device suitable for use with "time," but it is unlikely that this feature will be used much. However, it could be used to optionally allow operator changes between intervals of a program, with some default arranged if no reply is made within some time period.

The user can also extend the timing control of the simulator by using the multiprogrammer real-time clock, and the multiprogrammer wait instructions can be useful in special cases such as when several multiprogrammer instructions are passed in the same controller write instruction.

Bus Additions

The simulator can be expanded through the addition of new bus-compatible devices. Such devices can be given any available device addresses on bus 7 at first (see Table H-1). Up to three additional bus lines could be added to the controller's interface (if the interface slots are not used for some other purpose), but it is rather unlikely that this would be necessary to get more device addresses than are available on bus 7. Additional bus lines could be added if inter-

rupt capable measurement devices are added; by selecting a bus number higher or lower than 7, the user could control the interrupt priorities.

Lockouts

The controller may use the lockout (11o) command to prevent a device on the simulator's bus from being modified from the device's front panel (see Appendix H for a discussion of this in connection with the W175s). The user would generally lock out bus devices when they are being set and used by the controller, in order to prevent undetected status changes that could lead to faulty program results.

The user could also lock out the controller's live keyboard by disabling it (lkd), though this is not recommended because it would keep the operator from using the controller as a calculator during a test run, would prevent the use of the function keys, and would keep the operator from checking variable values. Disabling the keyboard may seem attractive as a way of keeping the operator from crashing a program by ill-advised keyboard changes, but if an operator wants to crash a program the operator can always find a way. It seems better to allow the operator free use of the live keyboard because it can improve the workload of a knowledgeable operator and disabling it would not keep an unknowledgeable one from finding a way to crash.

For/Next Loop Indices

It can make a user's program easier to read if a consistent use of variables is maintained throughout a test and between different tests. A suggestion in this line is that the for/next loops use the variables I and J as the main indices, with K, M, or N being used if more than two for/next loops are stacked (X and Y are used in the subroutine blocks). Q could be reserved for live keyboard for/next loops.

The user must avoid stacking for/next loops in such a way that an inner loop and an outer loop have the same variable as the index. This includes cases in which the inner loop is part of a subroutine called by the outer loop.

Flags

Flags 0-12 are freely available. These are useful for status indicators and branch or condition indicators, especially when such indicators are passed from one user subroutine or program segment to another.

HP6942 Outputs

It will be noted that most of the subroutine block multiprogrammer outputs are by single output instructions. This keeps each multiprogrammer output independent and increases the user's flexibility in combining subroutines. However, only a fraction of the multiprogrammer's power is used, since it can accept a number of instructions in one write output by the controller. It is difficult to use this power generally since the subroutine blocks (or other user-written independent subroutines) do not generally know in advance what combination of multiprogrammer instructions will follow. In a run type subroutine, the instructions are known but not necessarily the number of instructions or all of the instruction data; hence it is much easier to handle the output one instruction at a time. The user should keep the multiprogrammer's multiple instruction capability in mind and may find a use for it.

It should be noted that if multiple multiprogrammer instructions are sent from the controller in one write instruction, then the distinction between the multiprogrammer's serial mode (GS) and parallel mode (GP) becomes important (the multiprogrammer is in serial mode after "initial"). The HP6942 reference manual can be consulted for details and examples.

For/Next Line Breakup

If the HP9825's STOP key is pressed while the controller is running, the program will stop when the end of the current line is reached. There will be times when the operator will want to stop a for/next loop briefly and then go on with it by pressing CONTINUE. For example, during a run type output (such as a hoping noise spot), the operator may want to stop the run long enough to check the signal being changed (e.g., to look at the spot being hopped).

If a for/next loop is written so that it is entirely on one line, then that loop must complete before a STOP interrupt will take effect. It would be a good programming practice to break every for/next loop so that the loop is on at least two lines. An exception might be made when space is short and the loop is known to be short (few passes and little time per pass). An exception is also made for for/next loops run by the live keyboard, which must of course be one line. If stop is pressed during a live keyboard

for/next loop, the loop will run through to its end at a high rate regardless of any wait times written in the loop (such waits would be bypassed). A live keyboard for/next loop that is stopped by the operator cannot be continued after the stop.

"?" Position

The "?" subroutine block is used to check the multiprogrammer's busy instuction register (see Appendix O). It should be included in any user-written run type subroutines. Since it takes some time to execute this subroutine, it would be a good programming practice to put this subroutine in toward the end of a for/next loop rather than directly after the write instruction. Thus, the multiprogrammer instruction could complete while any other loop instructions (such as a wait) are carried out; then, when "?" is called it will not have to wait for the multiprogrammer (at least it would not have to wait as long). The arrangement of "AMown" (see Appendix Q), provides an example of positioning "?" properly.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX N

SUBROUTINE BLOCKS, ONE-LINE DESCRIPTION

1. ? (bit number, word number)
   Checks status of HP6942 busy instruction register and waits until instruction specified by parameters is complete.
2. fval # (VCO number, center frequency)
   Finds the D/A code number corresponding to the passed frequency and returns it in W.
3. fset (VCO number, center frequency)
   Sets the passed frequency on the specified VCO.
4. fnoise (VCO number, spot RF BW, video filter number/BW, band part)
   Sets in-channel noise of specified VCO. If spot not given, or given as zero, will turn off noise. Band part may be given as numbers 1-3 (low, mid, or high band) or as frequency (in the latter case, this will set that frequency as the center) or may default as mid band. Video filter may be given as a filter number or a bandwidth value. An explicit video number of zero will turn the noise generator off but leave the fill oscillator on to provide the requested spot.
5. pulse (VCO number, source number)
   Selects the modulation source switched into the pulse circuit.
6. biph (VCO number, source number)
   Selects the modulation source switched into the biphase circuit.
7. auxmod (VCO number, source number)
   Switches the coded modulation source through the auxiliary FM switch matrix to the specified VCO.
8. AMaux (VCO number, source number)
   Switches the coded modulation source through the auxiliary AM switch matrix to the specified VCO.
9. ampset (VCO number, dB attenuation)
   Sets the output power amplitude reference level in dB down from maximum output.
10. AMown (VCO number, rate, number points)
    Uses controller and required "AMval" user routine to run AM through D/A cards in HP6943.
11. setVCO (VCO number)
    Selects the active VCO in the appropriate RF channel.
12. initial
    Initializes the simulator status.
13. stepmod (VCO, number points)
    Uses controller and required "stepval" and "stepwt" user routines to run an arbitrary modulation pattern through the tune card.
14. ownswp (VCO number, low frequency, high frequency, rate, number sweep)
    Uses the controller to run a triangle D/A number sweep on the given VCO.
15. swp175 (VCO number, center frequency, frequency deviation, block rate, function number)
    Sets up the FM arbitrary waveform generator to provide a frequency deviation about the center, which is set on the appropriate tune card.
16. AM175 (VCO number, maximum dB attenuation, block rate, function number)
    Sets the AM arbitrary waveform generator to modulate the power amplitude of the VCO output.
17. DC175 (% duty cycle, period, W175 number, VCO numbers. . .)
    Sets up either of the arbitrary waveform generators as a pulse source for up to six VCOs.
18. T/P (period, VCO number. . .)
    Sets up the timer/pacer card as a pulse source for up to six VCOs.
19. special (band number, rate, running time, table length)
    Uses the controller and required "valspec" user routine to synchronously change the outputs of all VCOs in the band.
20. err stp
    Disables simulator when other subroutines detect an illegal condition.
21. shutoff
    Disables simulator when controller detects an error. Must be itself enabled to be in effect.
22. enter
    Subroutine function. Will convert an aphanumeric operator entry to numeric form.
23. inRFid
    Subroutine function. Will ask operator for RF number and VCO as A or B, and return VCO number.

24. loadW$ (VCO numbers. . .)
loadX$ (VCO number, video filter number, VCO number. . .)
loadY$ (VCO numbers. . .)
These subroutines will load data from the standard tape format to controller memory for W175, noise, and fill, respectively. Up to six VCOs (or VCO/filter number pairs) may be specified.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX O

SUBROUTINE BLOCKS, SHORT REFERENCE

This appendix briefly summarizes the main characteristics of each subroutine block. Included in each description are lists of the variables and data used, notes on the parameter ranges, and a brief description of the subroutine block's purpose. The form of each description is indicated in Table O-1. Two points should be noted about the descriptions. Under "variables used," the listed p-number is the highest numbered one used, and all lower p-numbers are also allocated. Flag 14 will be set and cleared by each subroutine block unless "no flag 14" is noted.

1. ? (bit number, word number)
   Size: 56 bytes
   Parameter ranges:
   bit number: 0-15 (integer)
   word number: 1, 2
   Subroutines used: none
   Variables used: p4; no flag 14
   Data used: none
   Error code: none

Table O-1 – Short reference form.

Label (parameter list)
Size: number of bytes
Parameter ranges:
Parameter number 1: range number 1
. .
. .
. .
Subroutines used : list
Variables used : list
Data used : list
Error codes : case number
cause number – cause
. .
. .
. .
Notes:
Short description of subroutine

The "?" subroutine block is used to avoid timing problems with the multiprogrammer (mp). The mp may have only one active instruction of a specific type at any time. Thus, for example, if an OS output instruction is being executed when a second OS instruction is sent from the controller to the mp, the second OS would be lost. This subroutine will read the mp busy instruction register until the instruction bit specified by the passed parameters is unset, indicating that the instruction has completed. "?" should be called whenever an output loop repeatedly uses the same mp instruction. It may also be called in nonloop cases as a general precaution. For easy reference, the bit and word numbers for the output serial (OS) and output parallel (OP) are:

OS: 4, 1
OP: 2, 1

2. fval# (VCO number, f_0)
   Size: 406 bytes
   Parameter ranges:
   VCO number: 1-12 (integer)
   f_0: f_{min} – f_{max}(Hz) (cf. Table C-2)
   Subroutines used: none
   Variables used: p9, V, W (return); no flag 14
   Data used: Z$
   Error code: 2
   number 0-f_0 less than f_{min}
   number 1-f_0 greater than f_{max}
   number 2-D/A number out of bounds

The "fval#" subroutine block is used to find the D/A converter number corresponding to the passed frequency and to return that number through the variable W. The D/A number is found by linear interpolation from the tune data held in Z$. "fval#" does not affect flag 14.

3. fset (VCO number, f_0)
   Size: 268 bytes
   Parameter ranges:
   VCO number: 1-12 (integer)
   f_0: f_{min} – f_{max}(Hz) (cf. Table C-2)
   Subroutines used: fval#, ?
   Variables used: p5, Z[*], W (indirect: V)

Data used: Z[*] (indirect: Z$)
Error code: 1
number 0 – illegal VCO number passed

The "fset" subroutine block is used to set the passed tune center frequency on the specified VCO.

4. fnoise (VCO number, spot RF BW, video number or BW, band part)
Size: 1316 bytes
Parameter ranges:
VCO number: 1-12 (integer)
spot BW: 0-f_{min}(Hz) (cf. Table C-2 and below)
video BW: 0-5 (integer) (cf. Table E-3)
or: (1K, 10K, 100K, 1M, 5M)(Hz)
band part: 1 (low), 2 (mid), 3 (high), or
f_{min} – f_{max}(Hz)
(if not given, defaults to mid band; see below)
Subroutines used: ?, fset (indirect: fval#)
Variable used: p16, Z[*], X (indirect: V, W)
Data used: X$, Y$, X[5], Z[*] (indirect: Z$)
Error code: 5
number 0 - illegal VCO number passed
number 1 - spot RF BW out of range
number 2 - illegal video filter passed
number 3 - band part frequency out of band
number 4 - spot RF BW cannot be achieved
(greater than 0 dB attenuation width)
Interior label: fnout

The "fnoise" subroutine block is used to set the passed noise spot on the specified VCO from the in-channel noise and fill sources. The noise spot is specified by the RF bandwidth, the video filter, and the part of the band in which the signal lies (band part).

The subroutine will turn off the noise spot if zero is passed as the spot RF BW. The same effect would be had if the subroutine were called with only the VCO number passed. Legal nonzero spot RF BW's must meet the following conditions:

$f_0 - \Delta f/2 <= fmin$
$f_0 + \Delta f/2 <= fmax$

Δf is identical with the spot RF BW, and f_0 is determined by the passed band part parameter (if this last parameter is not passed, it will default to mid-band). If band part is passed as a number (1, 2, or 3), f_0 will be:

$f_0 = [1.2 + 0.3(number - 1)]\ fmin$

If band part is passed as a frequency (any number not 1, 2, or 3), that frequency will be used as f_0. Moreover, in this last case the subroutine will call "fset" to set the tune center frequency.

Video noise filters 1 through 5 correspond to actual filters from 1 kHz to 5 MHz. A video filter of zero corresponds to turning off the noise generator while leaving the fill oscillator on; this is desirable when very small spots are wanted (though the resulting spot would have fill oscillator, not noise, characteristics). A video filter number of zero must be passed explicitly.

The software will check that a passed spot RF BW is greater than zero and less than the bandwidth of the VCO; the actual hardware limits will vary and will be more restrictive.

5. pulse (VCO number, source number)
Size: 232 bytes
Parameter ranges:
VCO number: 1-12 (integer)
source number: 0-7 (integer) (cf. Table C-1)
Subroutines used: ?
Variables used: p3, Z[*]
Data used: Z[*]
Error code: 3
number 0 - illegal VCO number passed
number 1 - illegal source number passed

The "pulse" subroutine block is used to select the modulation source connected to the passed VCO through the pulse modulation circuit. The source numbers and sources are:

0 - carrier on
1 - 10 Hz, 50% square wave
2 - 100 Hz, 50% square wave
3 - W175 (A)
4 - W175 (B)
5 - timer/pacer card
6 - external
7 - carrier off

6. biph (VCO number, source number)
Size: 230 bytes
Parameter ranges:
VCO number: 1-12 (integer)
Source number: 0-7 (integer) (cf. Table C-1)
Subroutines used: ?
Variables used: p3, Z[*]

Data used: Z[*]
Error code: 4
number 0 - illegal VCO number passed
number 1 - illegal source number passed

The "biph" subroutine block is used to select the modulation source connected to the passed VCO through the biphase modulation circuit. The source numbers and sources are:

0 - 20 MHz comb
1 - 10 MHz comb
2 - 5 MHz comb
3 - carrrier on
4 - 40 MHz biphase noise
5 - 20 MHz biphase noise
6 - 10 MHz biphase noise
7 - carrier off

7. auxmod (VCO number, source number)
Size: 300 bytes
Parameter ranges:
VCO number: 1-12 (integer)
source number: 0-3 (integer) (cf. Table C-1)
Subroutines used: ?
Variables used: p2, Z[12]
Data used: Z[12]
Error code: 6
number 0 - illegal VCO number passed
number 1 - illegal source number passed

The "auxmod" subroutine block is used to select the modulation source connected to the passed VCO through the FM auxiliary switch matrix. The source numbers and sources are:

0 - off
1 - external
2 - D/A FM
3 - W175 (B)

8. AMaux (VCO number, source number)
Size: 292 bytes
Parameter ranges:
VCO number: 1-12 (integer)
source number: 0-3 (integer) (cf. Table C-1)
Subroutines used: ?
Variables used: p2, Z[12]
Data used: Z[12]
Error code: 17
number 0 - illegal VCO number passed
number 1 - illegal source number passed

The "AMaux" subroutine block is used to select the modulation source connected to the passed VCO through the AM auxiliary switch matrix. The source numbers and sources are:

0 - off
1 - external
2 - D/A AM
3 - W175 (A)

9. ampset (VCO number, dB attenuation)
Size: 408 bytes
Parameter ranges:
VCO number: 1-12 (integer)
dB atten: 0-81
Subroutines used: ?
Variables used: p4, Z[11]
Data used: Z[11]
Error code: 7
number 0 - illegal VCO number passed
number 1 - passed dB value out of range

The "ampset" subroutine block is used to set the maximum output power amplitude level of the passed VCO. The subroutine sets this level in terms of dB below the absolute maximum, in 1 dB steps. Should a programmer wish to set the amplitude level in terms of dBm of output, the second passed parameter may be an expression:

X[7 + 2int((VCO # – 1)/6)
– (VCO #) mod 2] – (dBm value)

or

X(6 + band number) – (dBm value)

since attenuation (dB) = max. output power (dBm) – desired output power (dBm).

The "ampset" subroutine block, in practice, will usually be called to match the power levels of different VCOs, so that dB attenuation will be the usual unit of the second passed parameter.

10. AMown (VCO number, rate, number steps)
Size: 360 bytes
Parameter ranges:
VCO number: 1-12 (integer)
rate: $0.0305 - 10^3/X[4]$ [= 25] (steps/s)
number steps: ≥ 1
Subroutines used: ?, AMval, AMaux
Variables used: p7, U, X

Data used: X[*], Z[*]
Error code: 8
number 0 - illegal VCO number passed
number 1 - rate out of range
number 2 - "AMval" return out of bounds
(This subroutine ties up the controller.)

The "AMown" subroutine block is used to run a digital AM signal through one of the D/A cards in the multiprogrammer extender. The output rate and duration are determined by the parameters passed to this subroutine, while the output pattern (dB attenuation at any step) is determined by the return from "AMval." The "AMown" subroutine is thus a calling shell for the next value subroutine "AMval."

The controller is tied up when running this subroutine. The tie-up time is determined by the quotient of the passed number of steps divided by the passed rate (steps/s). Should a programmer wish to specify the AM pattern by time duration rather than by the number of AM steps, the third passed parameter can be an expression:

(time duration) * (rate) .

When the specified number of AM steps has been sent, "AMown" will restore the card word held in Z[*]. This will be identical with zero unless the user assigns some other value to the appropriate location in Z[*].

11. setVCO (VCO number)
Size: 186 bytes
Parameter range:
VCO number: 1-12 (integer)
Subroutine used: ?
Variables used: p2, Z[*]
Data used: Z[*]
Error code: 10
number 0 - illegal VCO number passed

The "set VCO" subroutine block is used to select the VCO to be used in the appropriate RF channel. The VCO numbers are given in Table C-2. Operator inputs of VCO number may be handled through "inRFid".

12. initial
Size: 400 bytes
No parameters
Subroutines used: ?
Variables used: X, Z[*]; no flag 14
Data used: none
No error code

The "initial" subroutine block is used to clear and initialize the hardware status of the simulator. The subroutine will connect VCO B in each RF channel, turn off the pulse and biphase carriers, set maximum power amplitude level attenuation, turn off the noise generators, and disconnect the FM and AM auxiliary switch matrices. The subroutine will also set the format of the multiprogrammer cards in slots 2 through 12 to 2's complement, and set the timer/pacer card to recirculating mode. The Z[*] array will also be initialized. The current calibration data identification line will be printed for reference.

13. stepmod (VCO number, number steps)
Size: 464 bytes
Parameter ranges:
VCO number: 1-12 (integer)
number steps: ≥ 1
Subroutines used: ?, stepval, stepwt
Variables used: p6, U, X, Z[*]
Data used: Z[*]
Error code: 11
number 0 - illegal VCO number passed
number 1 - "stepval" return out of bounds
number 2 - "stepwt" return out of bounds
(This subroutine ties up the controller.)

The "stepmod" subroutine block is used to run a stepped modulation pattern on the passed VCO. The step centers and step dwells are determined by the returns from "stepval" and "stepwt," respectively. The "stepmod" subroutine is thus a calling shell for the nextvalue subroutines "stepval" and "stepwt."

The controller is tied up when running this subroutine. The tie-up time is determined by the sum over the passed number of steps of the "stwpwt" returns. The subroutine will restore the initial tune center when it completes.

14. ownswp (VCO number, f_{low}, f_{high}, rate, number sweeps)
Size: 740 bytes
Parameter ranges:
VCO number: 1-12 (integer)

f_{low}: f_{min} to f_{max} (Hz) (cf. Table C-2)
f_{high}: f_{min} to f_{max} (Hz) (cf. Table C-2)
rate: see below (Hz/s)
number sweeps: ≥ 1
Subroutines used: ?, fval number
Variables used: p13, W, X (indirect: V)
Data used: X[3], Z[*] (indirect: Z$)
Error code: 9
number 0 - illegal VCO number passed
number 1 - dwell per sweep step too high
number 2 - dwell per sweep step too low
(This subroutine ties up the controller.)

The "ownswp" subroutine block is used to run a triangle frequency sweep on the passed VCO. The sweep parameters are determined by the passed parameters. The controller is tied up when running this subroutine. The tie-up time is determined by:

$$(\text{time}) = [2(f_{high} - f_{low})/(\text{rate})] * (\text{number sweeps}).$$

The subroutine runs a triangle that is linearly symmetrical in terms of the tune D/A number used. The actual frequency sweep will be asymmetrical since a real tuning curve will not be perfectly linear. The triangle introduces a multiplier of two into the controller tie-up time, as given above. A programmer may easily modify this subroutine to get a ramp sweep (multiplier of one), with either a positive or negative slope.

The valid parameter range for the passed rate depends on the passed sweep bandwidth and is related to restrictions on the controller dwell at each tune D/A number step. This dwell must be at least the controller program loop execution time in X[3] (corresponding to a minimum wait of zero) and at most the loop time plus about 33 s (corresponding to a maximum wait of 32767 ms). There will be a minimum of two output D/A numbers (going directly from one sweep limit to the other) and a maximum of 1 + D, where D is the difference between the high sweep limit D/A number and the low limit D/A number (going from one limit to the other by D/A steps of one).

If t = (dwell at D/A number) * (number of D/A numbers), then the valid parameter range for the passed rate is:

$$\Delta f/t_{max} \leq \text{rate} \leq \Delta f/t_{min} .$$

This can be worked out to give the joint restriction on the passed rate and passed sweep limits (including time-scale factors):

$$(2\times 10^3)(X[3]) \leq \Delta f/\text{rate} \leq (X[3] + 32767)(1 + D)(10^3) .$$

The "ownswp" subroutine will restore the initial tune center when it returns.

15. swp175 (VCO number, f_0, Δf, block rate, function number)
Size: 898 bytes
Parameter ranges:
VCO number: 1-12 (integer)
f_0: f_{min} to f_{max} (cf. Table C-2)
Δf: $f_0 + \Delta f/2 \leq f_{max}$;
$f_0 - \Delta f/2 \geq f_{min}$
block rate: 0 - 19.5 kHz (see below)
function number: 0-11 or 14-21 (integer)
Subroutines used: fset, auxmod (indirect: ?, fval#)
Variables used: p14, X, U$ (indirect: V, W, Z[*])
Data used: X[*], W$ (indirect: Z[*], Z$)
Error code: 14
number 0 - illegal VCO number passed
number 1 - block rate out of bounds
number 2 - illegal function number
number 3 - (f_0, Δf) combination out of bounds
number 4 - required W175 voltage out of range
(This subroutine will affect the display format setting.)

The "swp175" subroutine block is used to set up the FM arbitrary waveform generator (W175 (B)) to modulate the output of the passed VCO. The passed center frequency will be set on the tune card. The passed frequency deviation will be provided by the peak-to-peak voltage swing of the W175.

The deviation value set is the nominal value (at 19.5 kHz rate) based on the data table contents. The actual deviation bandwidth will vary somewhat with the W175's block rate. It is up to the programmer and operator to allow for this. The deviation value also depends on what part of the band the passed center is in; this is handled by the subroutine in deciding what part of its data table to use.

The form of the W175 modulation depends on the contents of the passed waveform function block, whether a full or partial block is used, and the block rate (or rather, sample time per point). The passed rate limits in this subroutine assume a full block is used; a programmer or operator wishing to use a partial block and a higher rate may use some suitable scaling expression when calling the subroutine.

The block rate can be related to the modulation sweep by:

$$\text{block rate} = (\text{sweep rate})/(k*\Delta f) .$$

Here k is some scale factor included to allow for block size and contents. For example, the scale factor for a full size block using function number 0 is 2. The

scale factor represents the number of maximum-minimum value swings in the function number contents.

The W175 voltage output set by this subroutine can be connected to other VCOs by using the "auxmod" subroutine. The resulting frequency deviation on the other VCOs can differ from that set on the VCO passed to "swp175," depending on the tune centers and tuning curves of the other VCOs (of course, if the other VCOs are in different bands, the frequency deviations will differ).

16. AM175 (VCO number, maximum dB attenuation, block rate, function number)
Size: 396 bytes
Parameter ranges:
VCO number: 1-12 (integer)
maximum dB: (see below)
block rate: 0-19.5 kHz (see below)
function number: 0-11 or 14-21 (integer)
Subroutines used: AMaux (indirect: ?)
Variables used: p5, U$ (indirect: Z[12])
Data used: X[*] (indirect: Z[12]
Error code: 18
number 0 - illegal VCO number passed
number 1 - block rate out of bounds
number 2 - illegal function number
number 3 - maximum dB value out of bounds

(This subroutine will affect the display format setting.)

The "AM175" subroutine block is used to set up the AM arbitrary waveform generator to modulate the output of the passed VCO. The form of the modulation depends on the contents of the selected W175 waveform block and the rate at which that block is used.

The rate passed through this block is the W175 block rate, and the range limits assume that a full block is used. Partial block usage at rates higher than 19.5 kHz requires that an appropriate scaling expression be used when passing the rate.

The maximum dB attenuation value passed sets the maximum signal attenuation due to the AM modulation circuit of the RF channel and should be understood as being relative to the power level attenuators ("ampset"). The parameter range for the passed dB value is related to the valid voltage range of the W175 and to the dB/V conversion factor of the D/A cards. The parameter range may be found as.

$$0.001\,X[11] \le (\text{dB value}) \le X[14]X[11]\ .$$

The usual range is currently (October 1981) estimated to be 0.0055 to 27.5 dB.

17. DC175 (% duty cycle, period, W175 number, VCO numbers. . .)
Size: 552 bytes
Parameter ranges:
% duty cycle: 0-100 (%)
Period: ≥ 0.051 (ms)
W175 number: 1 (W175-A) or 2 (W175-B)
VCO numbers: up to six numbers; must lie in different RF channels; 1-12 (integer)
Subroutines used: pulse (indirect: ?)
Variables used: p17, X, U$ (indirect: Z[*])
Data used: none (indirect: Z[*])
Error code: 15
number 0 - illegal VCO number passed
number 1 - attempt to use both VCOs in an RF head
number 2 - illegal W175 number passed
number 3 - % duty cycle out of range
number 4 - period out of bound

(This subroutine will affect the display format setting.)

The "DC175" subroutine block is used to set either of the arbitrary waveform generators to provide a pulsed square wave blinking signal to the pulse circuits of the passed VCOs. Up to six VCOs may be specified in the call, provided they lie in different RF channels. The blinking signal is defined in terms of the period and the duty cycle. This subroutine will affect the W175 block size; it will also turn off the 50 Ω output, leaving the 0 Ω output to be sent to the pulse circuits. Once set, the blinking signal may be connected to or disconnected from any RF channel by using "pulse."

The period (in milliseconds) may be replaced at the option of the user by an expression using rate (in Hz):

$$\text{period (ms)} = 10^3/\text{rate (Hz)}\ .$$

The user may also specify pulse width rather than period by using the expression:

$$\text{period} = 100^* \text{ pulse width}/\%\text{ duty cycle}.$$

18. T/P (period, VCO number. . .)
Size: 266 bytes
Parameter ranges:

period: $2 \times 10^{-}$ to 1×10^{7} (ms)
VCO numbers: up to six numbers; must lie in different RF channels; 1-12 (integer)
Subroutines used: pulse (indirect: ?)
Variables used: p15, X, Z[13] (indirect: Z[*])
Data used: none (indirect: Z[*])
Error code: 16
number 0 - period out of bound
number 1 - illegal VCO number passed
number 2 - attempt to use both VCOs in an RF head

The "T/P" subroutine block is used to set the timer/pacer card to provide a 50% duty cycle square wave blinking signal to the pulse circuits of the passed VCOs. Up to six VCO's (in different RF channels) may be passed in the same call. The subroutine assumes that the timer/pacer card is already in its recirculating mode. At the option of the user, rate may be passed instead of period, using the following relation:

$$\text{period (ms)} = 10^3/\text{rate (Hz)} .$$

The period will be held in Z[13]. Pulse width is one-half the period.

19. special (band number, rate, running time, table length)
Size: 392 bytes
Parameter ranges:
band number: 0-3 (integer) (cf. Table C-2)
rate: see below (Hz)
time: > 0 (s)
Table length: > 0
Subroutines used: ?, valspec
Variables used: p8, U, V, W, X, Y, Z; no flag 14
Data used: X[1]
Error code: 12
number 0 - illegal band number passed
number 1 - illegal table length passed
number 2 - rate out of bounds

(This subroutine ties up the controller.)

The "special" subroutine block is used to simultaneously change the tune centers and head function control words of all the available VCOs in the passed band. The new values are determined by the returns from "valspec." The dwell at each set of tune centers is determined by the passed rate, which describes the desired number of tune center changes per second. The parameter range for the rate is determined by the controller wait instruction limits and by the program loop execution time. The range may be expressed as:

$$10^3/(32767 + X[1]) \leq \text{rate} \leq 10^3/X[1] .$$

In practice, the valid range for the passed rate will be *0.05-27.3 Hz*.

The passed table length parameter is in turn passed to the "valspec" subroutine. It would normally be used to control the length of a table of values read by "valspec." This is necessary so that "valspec" can determine how much of the controller memory is given over to data. By varying the passed table length, a test may examine the effects of different repeat periods of the change pattern, with the longer periods coming closest to a random pattern. It should be noted, however, that the table length parameter could be used by "valspec" for other purposes, or indeed not used at all; the actual use is determined by "valspec." The controller is tied up when running this subroutine. The tie-up time (in seconds) is passed directly as the third subroutine parameter.

The subroutine resets the noise spots by sending out new channel function control words. In cases where the user wants one fixed control word per VCO (fixed spots), the variables Y, Z, and V can be assigned from Z[*] after setting the spots and before this subroutine is called; then "valspec" need only return the tune data. In other cases, the control word may vary. For example, by varying the noise spot according to the tune frequency, the user can get a pattern in which the apparent noise BW does not change with tune center. The user could also achieve a pattern in which a set of apparently fixed BW spots occasionally changes at the tune change step. In addition, biphase modulation may be switched on or off at each step.

20. err stp
Size: 286 bytes
No parameters
Subroutines used: none
Variables used: p24 (see below); no flag 14
Data used: Z
Error code: not applicable

The "err stp" subroutine block is used to disable the simulator when one of the other subroutines has detected a fault. This subroutine will turn off the pulse and biphase carriers and the noise of each of the RF channels. In doing so, it will reset each RF channel to VCO B. It will also turn off the 50 Ω outputs of the arbitrary waveform generators.

The subroutine will print out the error code information that was written into Z. It will use p21-p24 in manipulating Z; these p-numbers were selected to avoid overwriting the p-numbers of the block that entered "err stp." An operator could, therefore, read the contents of p-numbers as an aid in error tracing beyond the printed error code.

The subroutine ends in an endless loop, flashing a notice of the error. There is no explicit recovery from this subroutine; to exit, the operator must stop the machine, read the p-numbers if desired, and then reset the controller.

21. shutoff
    Size: 236 bytes
    No parameters
    Subroutines used: none
    Variables used: none (no flag 14)
    Data used: rom, ern, erl
    Error code: not applicable

The "shutoff" subroutine block is used to disable the simulator when the controller detects an execution error. The subroutine will turn off the pulse and biphase carriers and the noise of each of the RF channels. In doing so, it will reset each RF channel to VCO B. It will also turn off the 50 Ω outputs of the arbitrary waveform generators.

The subroutine will print out the error information available in the labels rom, ern, and erl. The subroutine then ends in an endless message loop from which there is no explicit recovery; the operator must stop and then reset the machine.

The subroutine must be enabled before it will take effect. If not enabled, an execution error will stop the program without affecting the output, which would be frozen at the state at which the error occurred. The enable statement, when used, should be among the first executable program lines. The subroutine is enabled by the following program line:

```
on err "shutoff"
```

22. enter
    Size: 490 bytes
    No parameters
    Subroutines used: none
    Variables used: p3; no flag 14
    Data used: U$
    Error code: none
    Interior label: →

The "enter" subroutine function block is used to convert an input entry string (U$) to numeric data form. It will allow data to be entered in terms of a multiplier unit, such as GHz or MHz. Syntax checking is not rigorous but should suffice.

The function assumes the desired input is contained in U$[1,32]. It will sequentially look for one of the multiplier characters it recognizes. The first one found (regardless of its position in the string) is taken to indicate the desired unit. The rest of the string, from the string beginning to just before the character position, is taken to contain numeric data. If this assumed numeric portion of the string contains non-numeric characters (other than blanks), the program will crash. If no recognized unit character is found, the entire string is assumed to be numeric.

Since the function only looks for one character, an operator only needs to enter that character. Thus, "G" is as equally acceptable as "GHz."

The characters currently recognized by the function are listed below, in the order in which the function looks for those characters. The function may readily be extended or modified to better match some particular program. For example, a program that does not need "milli-" units could remove the appropriate "m" line and modify the "M" line with the "cap" function. Use of this last function allows either the upper- or the lower-case character to indicate the units.

The characters currently recognized and their multipliers are:

g or G: giga-(10^9)
k or K: kilo-(10^3)
u or U: micro-(10^{-6})
M: mega-(10^6)
m: milli-(10^{-3})
d or D: decibels (1)
h or H: hertz (1)
s or S: seconds (1)
e: exponent (1)

23. inRFid
    Size: 224 bytes
    No parameters
    Subroutines used: none
    Variables used: p1; no flag 14
    Data used: U$
    Error code: none

The "inRFid" subroutine function block is used to accept an operator input specifying a VCO and convert it to the form used by the other subroutines. When used, this function will prompt an operator to specify a VCO in terms of the RF channel number and an identifying letter (A or B), which is the information an operator would find printed on the simulator's front panel. Input form is "#ℓ," where # is an integer 1-6 and ℓ is a, A, b, or B. Illegal inputs will be rejected and the operator reprompted. There is a default value of "6b," specifying VCO # 12.

24. load Y$ (VCO numbers. . .)
load X$ (VCO numbers, filter numbers. . .)
load W$ (VCO numbers. . .)
Size: 540 bytes
Parameter ranges:
VCO number: 1-12 (integer); up to six numbers; must lie in different RF channels; 1-12 (integer)
filter number: 1-5
Subroutines used: none
Variables used: p13 (Y$, W$); p20 (X$), X, V$[*]; no flag 14
Data used: Y$[*], X$[*], W$[*]
Error code: 13
number 0 - illegal VCO number, Y$
number 1 - attempt to use both VCOs in RF head, Y$
number 2 - illegal VCO number, X$
number 3 - illegal video filter number
number 4 - attempt to use both VCOs in RF channel, X$
number 5 - illegal VCO number, W$
number 6 - attempt to use both VCOs in RF channel W$

(These subroutines will set the tape track to track 1.) The "loadY$," "loadX$",and "loadW$" subroutine blocks are used to load data strings from tape to controller memory. The standard data tape format on track one is assumed. Up to six VCOs may be specified in a single call, provided all six lie in different RF channels. The video filter number passed in "loadX$" must be an integer 1-5, corresponding to a video filter value of 1 kHz to 5 MHz.

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX P

SUBROUTINE BLOCK, DETAILED DESCRIPTION

This appendix gives a detailed description of the subroutine block program instructions as of October 1981. The descriptions include what each program line does and what each local p-number is used for. The reader should refer to Appendix O for a short summary description of each subroutine block and to Appendix Q for a program listing.

When describing the lines in a subroutine block, this appendix will use relative line numbering (e.g., first line, second line, etc.) rather than absolute line numbers (e.g., line 0, line 9, etc.). This may seem slightly awkward; however, it keeps this appendix free of the actual line numbers, so that the descriptions could be applied to any listing regardless of the location of the subroutine blocks in memory and regardless of whether the listing for a subroutine block was from some subset of the blocks. Use of relative line numbers makes the description of each block independent of other descriptions. Hence one block could be modified by adding or cutting lines without throwing off the line references of all following descriptions.

For convenience, though, the subroutine blocks are described in the same order in which they would be found when reading the listing in Appendix Q. The actual line number in the listing of each subroutine block's first line is given as an aside to help the reader find that subroutine in Appendix Q; in keeping with the point of the previous paragraph, the reader is reminded that such actual line numbers may differ for other listings.

There are a number of program conventions and other common features that would be tedious to describe repeatedly. The reader should find it easier to follow the description if such common features are described separately; then they only need to be mentioned in the actual subroutine block description. For example, rather than describe the "err stp" branch in each subroutine, the branch can be described separately and the subroutine block description need only mention when the branch occurs. Common features are described below, with the subroutine block descriptions following.

The subroutine "?" is used by every subroutine that sends data to the multiprogrammer. The reader can assume this to be present between a subroutine's output to the multiprogrammer and the subroutine's return.

The subroutine blocks typically use the sort of structure indicated by Figs. P-1 and P-2, which show general flow charts. (Reference 2 includes a larger set of flow charts.) The general set type flow chart shows that a typical set type subroutine block will check the parameters passed to it, branching to "err stp" if a

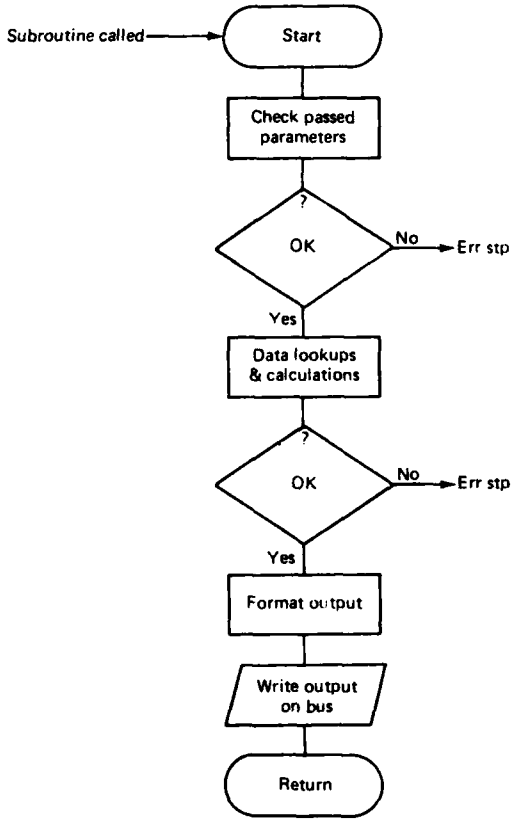


Figure P-1 — Set type subroutine block flow chart.

[2] J. M. Van Parys, *MMG ECM Simulator Software, Interim Documentation*, JHU/APL F4D-4-80(U)-002 (30 Sep 1980).

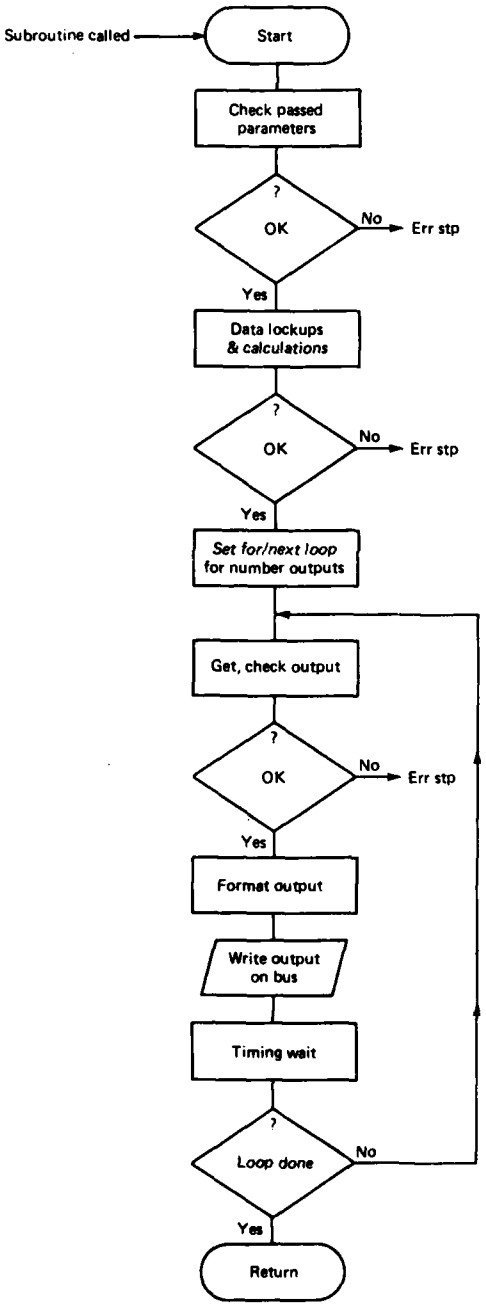


Figure P-2 — Run type subroutine block flow chart.

bad parameter is found. It would perform some data manipulations ranging from calculating a multiprogrammer slot number to searching a data table for entries used in an interpolation and may involve calling a separate subroutine. The results may be checked, with bad results causing an "err stp" branch. The properly formatted and addressed output would then be sent over the bus, and the subroutine would return.

The general run type flow chart shows that a typical run type subroutine is similar to a set type, up to the point where the output is found and sent over the bus. A run type subroutine will set up a for/next loop to handle a number of output steps. On each pass of the loop, the subroutine would find the next output value, typically by calling a nextval subroutine (see Appendix O). The output value may be checked with a bad value leading to an "err stp" branch. The properly formatted and addressed output would be sent over the bus and the controller would then wait for some amount of time in order to fix the output rate. The loop pattern would repeat until the specified number of output steps had been reached.

There are several exceptions to the structure suggested by the general flow charts, but most of the exceptions can be understood with the general flow charts if the reader mentally masks out part of a figure or mentally expands one step to include additional details. For example, the reader might mentally mask out the format and write steps in Fig. P-1; the flow chart should then serve to describe the structure of a calculation subroutine such as "fval#." The "?" subroutine block is one that cannot be modeled with the general flow charts; it is basically just a loop that waits for a bit in the multiprogrammer's busy instruction register to be unset before it returns.

The "err stp" branching noted in the general flow charts is a conditional branch to that label. The branch is activated if some conditional test indicates that a value is illegal or out of range. As part of the branch, the subroutine blocks will assign a coded number to the variable Z; this number will indicate the cause of the "err stp" branch that shut down the controller (see Appendix F and Table F-1). The form of an "err stp" branch is typically:

```
if (value) ( <, > ) (limit); (code) → Z;
    gto "err stp."
```

In the subroutine block descriptions, such a conditional test and branch to "err stp" will usually be indicated by noting a possible error branch.

The subroutine blocks will typically use a consistent pattern in passing parameters, such as giving a VCO number first, then a tune frequency, etc. (see Appendix C). Typically (there are exceptions, such as "?", "DC175", and "T/P"), the first passed parameter is a VCO number, so in reading the listings the reader can generally take p1 to be a VCO number.

Certain common calculations are used to find such things as multiprogrammer tune and function control slot numbers and the half of a tune card that should be used. These are indicated in Table C-2.

Tune D/A numbers and W175 voltages are found by simple linear interpolation from data in the appropriate tables. The particular table entries used are found by sequentially checking all appropriate table entries for the VCO involved until some condition (typically the desired value being greater or less than the tabled value) is met. This can be done since the tabled data are ordered (i.e., increasing attenuator setting, increasing W175 voltage, or increasing frequency). Because no table search involves checking more than eight entries, this simple easy-to-program approach is used; one of the theoretically more efficient table lookup algorithims would not give any noticeable improvement for such a short table search. On the noise, fill, and W175 tables, the search is narrowed down by specifying a band part number (see Appendix E).

The detailed descriptions of the subroutine blocks will use a common form to make it easier to pick out desired information. This form is:

"Label name" (passed parameters)
(Size in bytes; listing line number of first line)
Brief statement of what the subroutine does
Line descriptions
first line
second line
•
•
•

The first line is numbered 1, the second line is numbered 2, and so on. The reader is again reminded that these numbers are taken with reference to the subroutine and not to the actual line numbers of the subroutine block programming, and that the listing line number of the first line of the subroutine is for the particular listing in Appendix Q.

"?" (bit number, word number)
(56 bytes; line 0)

Checks multiprogrammer busy instruction register until instruction bit checked is unset.

1. Reads busy instruction register status words into p3 and p4. If passed bit of passed word is set, the program jumps to the beginning of the line.
2. Returns; reached when bit is not set (unset).

"fval#" (VCO number, f_0)
(406 bytes; line 2)

Finds a D/A number corresponding to the passed frequency by linear interpolation from the tune data.

1. Scales passed frequency to gigahertz and checks that it is not less than the minimum tabled frequency in Z$ (note that the VCO number is not checked; this will normally be done by another subroutine calling this one).
2. Sets up a for/next loop (index V) to find the first table entry less than or equal to the scaled frequency. If such an entry is found, the entry number is saved in p4 and the loop ended by manipulating the index within the loop.
3. Checks that a table entry was found in the second line; otherwise the passed frequency was too high and an error branch is set.
4. Sets low and high data table entry points in p4 and p9, respectively. Gets the high D/A number and the difference between high and low D/A numbers and saves them in p6 and p5, respectively.

Note: Low and high refer to the entry numbers; low frequency will be less than high frequency but "low" D/A number may or may not be less than "high" D/A number.

Note: If the linear interpolation is thought of in terms of X and Y, the D/A number corresponds to the Y axis.

5. Gets the high frequency (p7) and the high-low frequency difference; then calculates the local slope of the Z$ data about the entry points. The slope is saved in p5.
6. Calculates the desired D/A number corresponding to the passed frequency and assigns it to W. Checks that the D/A number is within the 8 bit absolute D/A range.
7. Returns.

"fset" (VCO number, f_0)
(268 bytes; line 9)

Sets the passed frequency as the tune center for the specified VCO. This calls "fval#".

1. Sets flag 14, then checks that the VCO number is legal.
2. Calculates the RF number (p3) and finds the binary shift and mask bytes (p4 and p5, respectively).
3. Calls "fval#" to get the D/A number, then manipulates the D/A number to fit into the proper half of Z [*].
4. Sends the new Z[*] word to the multiprogrammer, clears flag 14, and returns.

"fnoise" (VCO number, spot RF BW, video, band part)

(1316 bytes; line 13)

Sets a noise spot on the output from the specified VCO, using the in-channel noise generator and fill oscillator. Band part may be a number 1 (low part of band), 2 (mid), or 3 (high); it may be a frequency, or it may be defaulted (not given) to midband. The spot can be turned off by calling the subroutine with just the VCO number given or by passing a value of zero for the spot RF bandwidth. This subroutine may call "fset".

1. Checks that the VCO number is legal.
2. Calculates the RF number (p5) and the channel function control slot number (p10). The minimum frequency is calculated from the band number and saved in p11.
3. Checks if the spot RF BW is zero (explicitly passed zero or only the VCO number passed); if it is, the fill attenuation (p13) and noise attenuation (p14) values are set and the subroutine branches to its "fnout" label.
4. Checks if the video was specified by a video filter number 0-6; if so, that number is assigned to p6 and the subroutine jumps three lines to the seventh line.
5. Checks if the video was specified as 5 MHz; if so, sets the video number (p6) and jumps two lines to the seventh line.
6. Calculates the video number (p6) from the video bandwidth and checks that it (and hence the passed video bandwidth) is legal.
7. Uses the passed band part parameter as an initial value for the reference center frequency (p13). Checks if the band part was actually passed; if not, it will set a midband frequency as the reference center (p13) and will set the data table band part offset (p7), then jump four lines to the eleventh line.
8. Checks if the band part was specified by a low/mid/high code number; if so, it sets reference center (p13) and table offset (p7) values, then jumps three lines to the eleventh line.
9. If this line is reached, it is assumed the band part was specified by passing a frequency. This line checks that the value is within the band limits, the high limit being assumed equal to twice the low limit.
10. Calls "fset" to set the passed band part frequency as the tune center, then finds the data table band part offset (p7).
11. Checks that the desired spot bandwidth about the reference center will not be clipped, by checking if the spot would overlap the band limits.
12. Divides the desired spot RF BW into fractions from the fill oscillator (p8) and the noise generator (p9) and initializes full attenuation for the fill attenuator (p13) and noise attenuator (p14) settings. If the video was specified as zero, it will reset the fill and noise spots so that all of the desired spot RF BW will come from the fill oscillator.
13. Checks that the fill oscillator spot is not greater than the zero attenuation fill oscillator bandwidth. If the spot is greater, then the fill spot is reset to the zero attenuation value, the fill attenuation value set for zero attenuation, the noise generator spot adjusted to maintain the overall spot bandwidth, and a flag (p15) set to indicate that this resetting has been done.

Note: This check may be unnecessary or undesirable in practice. If the user prefers, the conditional action after the check could be changed from the resetting to an "err stp" branch; line 14 should be modified to match, and lines 15, 16, and 19 could be cut.

14. Similar to the thirteenth line, but for the noise generator spot. The flag is p16. See the note above.
15. Checks if both resetting flags (thirteenth and fourteenth lines) have been set.
16. Checks if the fill spot has been reset; if so, the program jumps three lines to the nineteenth line.
17. Sets up a for/next loop (index X) to search the fill oscillator data for the first tabled entry less than the desired fill spot. If such an entry is found, the previous index is used for the attenuator setting (i.e., if the table entry for an attenuator setting of 4 is found to be less than the desired spot, then the previous attenuator setting, in this case a setting of 3, is used as

being the closest setting greater than or equal to the desired spot). If no table entry is found, then the full attenuator setting that was set in the twelfth line is used.

Note: The table search may be easily modified if actual use of the simulator suggests a better approach. The check could be for "less than or equal to" and the present index used for the attenuator setting (replace X-1 by X). The check could also be modified to branch to "err stp" if the full attenuation value is too large (see second and third lines of "fval#").

18. Completes the fill oscillator data search.
19. Checks if the noise generator spot has been reset; if so, the program jumps three lines to the twenty-second line.
20. Similar to line 17, but for the noise generator attenuation. The note after line 17 also applies to this line.
21. Completes the noise generator data search.
22. "fnout" label. Sets flag 14 and masks out the proper Z[*] card word.
23. Forms the Z[*] channel function control word so it contains the new fill oscillator and noise generator attenuator settings and the new video number.
24. Sends the new Z[*] word to the multiprogrammer, clears flag 14, and returns.

"pulse" (VCO number, source number)

(232 bytes; line 37)

Sets the pulse circuit of the RF channel of the specified VCO so as to connect the specified source (see Table C-1).

1. Checks that the VCO number is legal.
2. Checks that the source number is legal.
3. Sets flag 14 and forms the new Z[*] head function control card word.
4. Sends the new Z[*] word to the multiprogrammer, clears flag 14, and returns.

"biph" (VCO number, source number)

(230 bytes; line 41)

Sets the biphase circuit of the RF channel of the specified VCO to provide the specified biphase modulation (see Table C-1). This subroutine is similar to "pulse" (immediately above), differing only in the values of the shift and mask bytes. The line-by-line description of "pulse" can be directly used for this subroutine as well.

"auxmod" (VCO number, source number)

(300 bytes; line 45)

Sets the auxiliary FM switch matrix to connect the specified source to the RF channel of the specified VCO.

1. Checks that the VCO number is legal.
2. Checks that the source number is legal.
3. Sets flag 14, then forms the new Z[12] auxiliary switch word to represent the new status of the matrix, including the RF channel involved.
4. Sends the new Z[12] word to the multiprogrammer and waits a minimal time for that value to settle.
5. Disables the switch matrix from following changes until this subroutine is called again, clears flag 14, and returns.

"AMaux" (VCO number, source number)

(292 bytes; line 50)

Sets the AM auxiliary switch matrix to connect the specified source to the RF channel of the specified VCO. This subroutine is virtually identical to "auxmod" (immediately above), differing only in the shift and mask bytes, and the same line-by-line description used for "auxmod" also describes "AMaux."

"ampset" (VCO number, dB attenuation)

(408 bytes; line 55)

Sets the output power reference level by specifying the dB of attenuation (0-81 in 1 dB steps) set by the level set attenuators.

1. Checks that the VCO number is legal.
2. Assigns the passed dB value into a local variable (p4) so that a later instruction (the seventh line) need not depend on some conditional assignments (third and fourth lines). This line also checks that the passed dB value is legal.
3. Checks if the passed dB value is 81 dB; if so, it sets the coarse (p4) and fine (p3) attenuator codes, then jumps three lines to the sixth line.
4. Similar to the third line, except that it checks for a passed value of 80 dB.
5. Sets the fine attenuation code (p3); p5 is used as a temporary holding variable.
6. Sets flag 14 and masks out the previous power level data from Z[11].
7. Forms the new Z[11] word to contain the RF

channel number (offset by −1 to match the hardware), the fine attenuation code, and the coarse attenuation code.

8. Sends the new Z[11] word to the multiprogrammer, clears flag 14, and returns.

"AMown" (VCO number, rate, number of output steps)

(360 bytes; line 63)

Uses the controller to run an AM pattern through one of the AM D/A cards in the multiprogrammer extender. The pattern is determined by the user-provided nextval subroutine "AMval" (see Appendix D). This also calls "AMaux".

1. Checks that the VCO number is legal.
2. Calculates the wait time (p4) and checks that it is within range.
3. Calculates the multiprogrammer extender slot number of the AM D/A card for the RF channel of the specified VCO (p5), then calls "AMaux" to connect the RF channel and D/A card. The line then sets up a for/next loop (index X) to run the output pattern, starting by calling the nextval "AMval".
4. Checks that the "AMval" return in U of a dB value is within range.
5. Sends the value in U to the multiprogrammer, waits to establish the rate, and continues the for/next loop.
6. Restores the D/A card to the value held in Z[*] before the output pattern was run (unless the user directly assigns a new value to Z[*] after "initial", this will amount to setting the D/A card to 0 dB attenuation when the output loop is done). The line then returns the subroutine to the calling program.

"setVCO" (VCO)

(186 bytes; line 69)

Sets the active VCO of the pair in each RF channel.

1. Sets flag 14, then checks that the VCO number is legal.
2. Calculates the channel function control card slot number (p2) and forms the new Z[*] word. An even VCO number ("B" labeled VCO of an RF channel pair) gives a control bit of 0.
3. Sends the new Z[*] word to the multiprogrammer, clears flag 14, and returns.

"initial"

(400 bytes; line 72)

Sets the initial status of the simulator (see Appendixes B and J, and Table J-1). This will set the tape to track 1. Z[*] and U$ must be dimensioned before calling this.

1. Clears the bus, gets and prints the calibration data identification line (see Appendix E), waits to allow the multiprogrammer to pass self-test, and sets the timer/pacer card to its recirculating mode.
2. Waits to allow the "WF" instruction of the last line to complete ("?" cannot be used since "WF" is not monitored by the busy instruction register), then initializes Z[*]. A for/next loop (index X) is set up to change the data format of the digital output cards in slots 2-12 to 2's complement. A brief wait allows each instruction to complete before the next is sent (it was found in practice that "?" would not work if used when setting the data formats).
3. Completes the loop begun in line 2. A new for/next loop is begun to initialize the status of the RF channels. It first sets the channel function control cards and the corresponding Z[*] words.
4. As part of the loop begun in line 3, the VCO select bit is flip-flopped to ensure that it will be in a known state.

Note: The fourth line of "initial" is an example within the subroutine blocks (however trivial) of use of the multiprogrammer's capability for accepting several instructions at the same time.

5. Continuing the loop begun in the third line, this line sets the auxiliary FM and AM switch matrices to off states.
6. Sets the level set attenuators to provide full output power attenuation, then continues the loop begun in the third line.
7. Disables the auxiliary switch matrices from following changes, then returns.

"stepmod" (VCO number, number of output steps)

(464 bytes; line 79)

Uses the controller to run an arbitrary output pattern involving the specified VCO. The pattern is determined by user-provided nextval subroutines "stepval" and "stepwt" (see Appendix D).

1. Checks that the VCO number is legal.
2. Calculates the tune card slot number (p4).

3. Sets up p6 for local use in place of the tune word in Z[*]; then finds the shift (p5) and mask (p3) bytes.
4. Sets flag 14, sets up a for/next loop (index X) to run the output pattern, and calls the nextval "stepval".
5. Checks that the "stepval" return in U of a D/A number is within the absolute 8 bit range for that number.

Note: The D/A number range that gives meaningful frequency outputs will generally be considerably less than 0-255; however, the absolute range is easier, quicker, and shorter to check than the proper limits held in the appropriate parts of Z$.

6. Forms and sends to the multiprogrammer the new tune word.
7. Calls the nextval "stepwt" to get the total dwell until the next output change, offsets the return in U of a dwell in milliseconds by the loop time to get the wait, and checks that the wait is within range.
8. Waits to establish the dwell, then continues the output loop.
9. Restores the original pre-"stepmod" tune data, clears flag 14, and returns.

"ownswp" (VCO number, low frequency, high frequency, rate, number of sweeps)

Uses the controller to run a frequency sweep by changing the tune center of the specified VCO. A triangle waveform is used to change the tune card D/A numbers; the frequency may be nonlinear and nonsymmetrical due to nonlinearities and hysteresis effects in the tuning curve. This calls "fval#".

1. Checks that the VCO number is legal.
2. Calculates that tune card slot address (p10), and the shift (p14) and mask (p12) bytes.
3. Masks out the tune word in Z[*] and assigns it to p15 for local use. The line then calls "fval#" to get the low frequency D/A number, saving it in p7.
4. Calls "fval#" to get the high frequency D/A number, saving it in p8; then, if necessary, will swap p7 and p8 (using p11 as a holding variable) so that the low D/A number in p7 is really less than the high D/A number in p8.
5. Calculates the time allowed for one sweep (in milliseconds) from the frequency limits and the rate, saving this time in p11. The line also gets the difference between the high and low D/A numbers, saving this in p6. The quotient of the time for one sweep divided by the D/A number difference is offset by the loop time and saved in p13. This gives the wait time that would be needed if the controller swept between frequency limits with a D/A number change per step of one. Such a sweep would give the best resolution and also would be used to get the slowest possible sweep. The wait time is checked to see if it is too large (i.e., if the rate is too low for the specified frequency sweep range, or alternately, if the frequency sweep range is too large for the specified rate).

Note: The line should also check that the value in p13 is not less than zero, which would indirectly check either the order of the low and high frequency limits or the sign of the rate (but not both simultaneously; if the low and high limits are reversed and the rate is negative, p13 would be positive). The line would have to be split in two for the zero check to be made, however.

Note: If the user does need a controller run sweep in which each step needs more than 32767 ms, the user can modify the wait method to get any wait greater than zero (e.g., a for/next loop repeating a number of waits, or use of the multiprogrammer clock).

6. Sets the D/A number change per output step (p9) as 1. If the wait time in p13 is greater than or equal to zero, this is the combination of D/A number change per step and wait per step that will be used, and the program jumps to the eighth line.
7. If this line is reached, then a D/A number change per output step of one would take too long. The wait per step is reset as zero, and the line calculates the D/A number change that would then be needed. This change would typically be greater than one, giving less resolution but taking less time to run. The line checks that the D/A number change is achievable (an error here would imply that the rate is too high for the specified frequency limits, or alternately, that the frequency limits are too low for the specified rate).
8. Sets up a for/next loop (index Y) to run the specified number of sweeps, then sets up an inner for/next loop (index X) to run the up slope of the D/A number triangle waveform.

Note: The user can easily modify this subroutine to use a ramp waveform to run the D/A number

sweep by cutting out one or the other of the inner (X index) loops. The user is reminded that up and down refer to the sweep of the D/A numbers and this may or may not correspond to the sense of the resulting frequency sweep.

9. Handles the word formatting, multiprogrammer output, and wait time for each step of the up slope.
10. Sets up an inner for/next loop to handle the formatting and multiprogrammer output for each step of the down slope.
11. Waits to establish the rate of the down slope, then continues with the output sweep loop. When the output sweep loop is done, the original presweep tune data are restored.
12. Clears flag 14 and returns.

"swp175" (VCO number, center frequency, frequency deviation, W175 block rate, function number)

(898 bytes; line 100)

Sets up the FM W175 and tune center to give an FM waveform as specified by the passed parameters. This calls "fset" and "auxmod".

1. Checks that the VCO number is legal.
2. Calculates the RF number (p6), then checks that the block rate is legal (see Appendix H).
3. Checks that the function number is legal.
4. Calculates the band number (p7) and initializes the data table pointer (p12).
5. Calculates a minimum frequency based on the band of the specified VCO (p10) and finds the data table part offset (p11).
6. Checks that the frequency deviation about the center will be within range.
7. Sets up a for/next loop (index X) to search the W$ data table for the first entry greater than or equal to the desired deviation bandwidth. If such an entry is found, then the table number of the entry is saved in p12 and the search ended by manipulating the for/next index within the loop. The value of the data table entry will be in p14; this is the high voltage.

Note: If the voltage-bandwidth calibration curve is thought of in terms of X and Y, voltage is the Y axis.

8. Completes the table search. If no entry was found in the search, the subroutine will use the last entry (highest tabled voltage).
9. Finds a low voltage (p13) to be used with the high voltage in p14.
10. Calculates the voltage (p8) corresponding to the specified frequency deviation bandwidth. The local slope of the calibration curve is in p13.
11. Checks that the voltage calculated in the previous line is legal.
12. Sets up U$ to contain the specified block rate.
13. Modifies U$ to also specify the voltage amplitude and offset and the function number, and also to turn on the 50 ohm output.
14. Calls "fset" to get the specified tune frequency, sends U$ to the FM W175 to get the FM, calls "auxmod" to connect the FM W175 and the RF channel of the specified VCO, then returns.

"AM175" (VCO number, maximum dB, W175 block rate, function number)

(396 bytes; line 114)

Sets up the AM W175 to give an AM waveform as specified by the passed parameters. This calls "AMaux."

1. Checks that the VCO number is legal.
2. Checks that the block rate is legal (see Appendix H).
3. Checks that the function number is legal.
4. Calculates the W175 voltage corresponding to the specified maximum dB depth (p5) and checks that it is within range.
5. Sets up U$ to contain the specified block rate.
6. Modifies U$ to contain the voltage amplitude and offset and the function number, and to turn on the 50 ohm output.
7. Sends U$ to the AM W175 to set the AM waveform, calls "AMaux" to connect the AM W175 to the RF channel of the specified VCO, and returns.

"DC175" (% duty cycle, period, W175 number, VCO numbers. . .)

(550 bytes; line 121)

Sets up either W175 to provide a simple pulse waveform to the RF channels of the specified VCOs. Up to six VCOs may be specified if they are all in different RF channels. The AM W175 is indicated by a W175 number of 1 and the FM W175 is indicated by a 2. The simple pulse waveform (one on/off pulse per period) is taken from the W175 square wave function block; a partial block is used to get the specified duty cycle. The period in milliseconds is used to set the block rate in hertz.

1. Sets up a for/next loop (index X) to check the passed VCO numbers, and checks that the VCO numbers are legal.
2. Calculates the RF number (p11) of each VCO number and checks that RF number has not already been used.
3. Sets a flag (p12-p17) to indicate that the RF number has been used, and continues the for/next loop. The line then checks if the W175 number is legal.
4. Checks if the % duty cycle is within range.
5. Checks if the period is too low.

Note: The fifth line checks the period against a limit of 0.051 (ms), corresponding to a maximum rate of 19.5 kHz, the full block limit. Since a partial block will be used (unless the duty cycle is 50%), the actual limit would be lower, depending on the actual block size and hence the duty cycle. The limit in the fifth line might be replaced by an expresion that allows for the actual block size, which would be somewhat more complex. The limit value might also be replaced by a simple value of 2e-3 for the 5.1e-2, to reflect the 500 kHz response limit of the pulse circuits. There is no upper limit check since it is felt unlikely that any real test will need pulse periods greater than 71 hours.

6. Calculates the start and stop addresses (p13 and p12, respectively) for the square wave function block. The line modifies the stop address if the duty cycle is less than or equal to 50%.
7. Modifies the start address if the duty cycle is greater than 50%.
8. Sets U$ to contain the start and stop addresses, the partial block indicator, the voltage amplitude and offset, and the function block number, and to turn off the 50 ohm output.
9. Modifies U$ to contain the block rate corresponding to the passed period; then conditionally jumps one or two lines, depending on which W175 was specified as the pulse source.
10. Sends U$ to the AM W175, then jumps two lines to the twelfth line.
11. Sends U$ to the FM W175.
12. Sets up a for/next loop (index X) that calls "pulse" to connect the W175 serving as the pulse source with the RF channel of each passed VCO.
13. Completes the for/next loop, then returns.

"T/P" (period, VCO numbers. . .)

(266 bytes; line 134)

Sets the multiprogrammer timer/pacer card as a pulse waveform source for the specified VCOs. Up to six VCOs may be specified if they lie in different RF channels. The pulse waveform is a 50% duty cycle square wave. The timer/pacer card should already be in its recirculating mode, as set by "initial"; if the user has changed the card mode it is up to the user to reverse that change.

1. Checks that the period is legal.
2. Sends the period to the multiprogrammer, putting the new period value in Z[13].
3. Sets up a for/next loop to handle the checking and setting for each passed VCO number. The loop begins by checking that the VCO number is legal.
4. Calculates the RF number of the VCO being checked (p9) and checks that the RF channel has not already been used.
5. Sets an RF use flag (p10-p15), then calls "pulse" to connect the timer/pacer card to the pulse circuit of the RF channel of the specified VCO. The loop then continues. When done, the subroutine returns.

"special" (band number, rate, time at rate, table length)

(392 bytes; line 139)

Uses the controller to run a synchronous pattern on the three available VCOs in a band. Both tune centers and channel function control words can be changed at each step. The changed values are determined by the user-provided nextval function "valspec" (see Appendix P). The reader should see the notes on this subroutine in Appendix K.

1. Checks that the band number is legal.
2. Determines the channel function control card slot numbers for the three VCOs in the band (p6, p7, and p8, in order of increasing VCO number) and the slot number for the double word tune card (i.e., the tune card on which both halves control a VCO within the specified band).
3. Checks that the table length parameter is not less than zero.
4. Calculates the wait on each output step (p9) and checks that it is legal.
5. Calculates the number of output steps (p7), then sets up a for/next loop (index X) to handle the output. The loop first calls "valspec."

6. The unchecked and unmanipulated returns from "valspec" are sent directly to the multiprogrammer, the controller waits to establish the rate, and the output loop continues until done, when the subroutine returns.

"err stp"

(286 bytes; line 145)

Handles the simulator shutdown and program stop directed if one of the subroutine blocks finds an error during a check (see Appendix F). This is not a true subroutine since it is entered by a branch rather than a call, and it has no return.

1. Sets the display format and turns off the 50 ohm outputs of the W175s.
2. Sets the channel function control cards to their "initial" state: pulse and biphase carriers turned off, fill oscillator and noise generator set to full attenuation, and video turned off. The VCO select will be set to the "B" labeled VCO in this process. It may be noted that this does not affect the Z[*] contents.
3. Formats the error code contents in Z into three parts (p21, p22, and p23; p24 is a holding variable). See Table F-1 for a list of the subroutine block error codes.
4. Writes the formatted error code on the HP9825's internal printer.
5. Sets up an endless loop that beeps and flashes an operator's notice of the error.

"shutoff"

(236 bytes; line 150)

Handles the simulator shutdown and program halt directed if a controller error is encountered. This must be enabled (on err "shutoff") before it takes effect. It is strongly similar to "err stp", differing only in the error information printed.

1. Prints a notice on the HP9825's internal printer, turns off the 50 ohm outputs of the W175s, and sets the display format.
2. Turns off the simulator through the channel function control cards (see the second line of "err stp").
3. Writes the controller's error information on the HP9825's internal printer.
4. Sets up an endless loop that beeps and flashes an operator's notice of the fault.

"enter"

(490 bytes; line 154)

Converts and scales an input alphanumeric string to a numeric form that is returned by this subroutine function (see Appendix C). The syntax checking is minimal but should work. This function will sequentially check for a recognized character anywhere in U$. Note that this means the operator cannot combine an exponential form number with a character. If a recognized character is found, the function will treat all of the string up to that character as the numerical part of the string. Each line is very similar, with p1 as the scale multiplier, p2 as the end point of the numeric portion of U$, and p3 as a character position holding variable. If a character is recognized by a line, that line will branch to the last subroutine function line.

1. Initializes the multiplier (p1) and end pointer (p2), then checks for the giga prefix.
2. Checks for the kilo prefix.
3. Checks for the micro prefix.
4. Checks for the mega prefix.
5. Checks for the milli prefix.

Note: As mentioned in Appendix K, these last two lines could be modified to eliminate the milli check and allow either an upper- or a lower-case letter M to indicate the mega prefix.

6. Checks for a dB indicator.
7. Checks for a Hz indicator.
8. Checks for a seconds indicator.
9. Checks for the exponent sign (if found, then the entire string is assumed numeric).
10. "→" label. Finds and returns the scaled numeric content of the string.

"inRFid"

(224 bytes; line 164)

Prompts an operator to specify a VCO by the RF channel number and the VCO letter label, information available to an operator looking at the front of the simulator rack, and finds the VCO number of the specified VCO. This is a subroutine function.

1. Clears out old U$ contents and specifies a default of 6B (VCO number 12), then prompts the operator to make an entry. The cap function is used so that the operator may use upper- or lower-case letters.
2. Checks that the RF channel number is legal; if it is, the program jumps two lines to the fourth line.
3. Notifies the operator that the entry form was bad, indicates the correct form, and then jumps back two lines to the first line.
4. Checks that the VCO letter label is legal; if

not, the program jumps back one line to the third line (and thence back to the first).

5. Calculates and returns the VCO number of the specified VCO.

"load Y$" (VCO numbers. . .)
"load X$" (VCO numbers, video number, . . .)
"load W$" (VCO numbers. . .)

(540 bytes; line 169)

These subroutines are similar enough that they can be described together as a general "load$". They are used to update the controller data from tape. They are separate so that only the data affected by some program change need be updated. Fill oscillator and FM W175 data are updated by "loadY$" and "loadW$", respectively; the only difference between the two is the string involved and the tape files used to get the updated data. Noise generator data are updated by "loadX$", which includes a line to identify the video filter number. Up to six VCOs (or six VCO/video number pairs for "loadX$") may be specified in one call, provided the VCOs all lie in different RF channels.

The following line-by-line description applies equally well to either "loadY$" or "loadW$"; the description for "loadX$" that then follows refers to the earlier description.

1. Sets the tape track number to track 1, then sets up a for/next loop (index X) to handle each passed VCO.
2. Checks that the VCO number is legal.
3. Calculates the RF channel number (p13) and checks that that channel has not already been used.
4. Sets an RF use flag (p7-p12), then transfers the taped data through V$ to the controller data string.

The line-by-line description for "loadX$":

1. Same as first line above.
2. Same as second line above.
3. Checks that the video number is legal and nonzero, using p20 as a holding variable.
4. Same as third line above (except that p19 is used to hold the RF number).
5. Same as fourth line above (except that p13-p19 are used as the RF use flags).

THE JOHNS HOPKINS UNIVERSITY
APPLIED PHYSICS LABORATORY
LAUREL MARYLAND

APPENDIX Q

SUBROUTINE BLOCKS, PROGRAM LIST

Note: The type face of the printer used for the appended program listing differs from that of the HP9825's internal printer. The assignment arrow has been replaced by a right parenthesis and the power arrow by a caret.

Note: The program listing is that of October 1981.

```
0: "?":red 72313,p3,p4;if bit(p1,p(2+p2));jmp 0
1: ret
2: "fval#":if (p2/1e9)p3)<val(Z$[p1,1,5]);2e3+p1)Z;gto "err stp"
3: for V=2 to 6;if p3<=val(Z$[p1,9V-8,9V-4]);V)p4;10)V
4: next V;if V<10;2100+p1)Z;gto "err stp"
5: (9p4)p9)-9)p4;(val(Z$[p1,p9-3,p9])}p6)-val(Z$[p1,p4-3,p4])}p5
6: p5/((val(Z$[p1,p9-8,p9-4])}p7)-val(Z$[p1,p4-8,p4-4])}p8)}p5
7: int(p5(p3-p7)+p6)}W;if W<0 or W>255;2200+p1)Z;gto "err stp"
8: ret
9: "fset":sfq 14;if p1<0 or p1>12 or frc(p1);1e3)Z;gto "err stp"
10: 2+int((p1-.5)/4)}p3;-8}p4;255}p5;if int(p1mod4/2+.5)=1;0}p4;65280}p5
11: cll 'fval#'(p1,p2);ior(band(Z[p3],p5),shf(W,p4))}Z[p3]
12: wrt 723,"OS",p3,Z[p3],"T":cll '?'(4,1);cfq 14;ret
13: "fnoise":if p1<0 or p1>12 or frc(p1);5e3)Z;gto "err stp"
14: (int(p1/2+.5)}p5)+4}p10;(2.5e8)2^(1+2int((p1-1)/6)-p1mod2))p11
15: if p2=0;7}p13}p14;gto "fnout"
16: if p3>=0 and p3<6 and not frc(p3);p3)p6;jmp 3
17: if p3=5e6;5}p6;jmp 2
18: log(p3)-2}p6;if p6<1 or p6>4 or frc(p6);5200+p1)Z;gto "err stp"
19: p4}p13;if p0=3;40}p7;1.5p11}p13;jmp 4
20: if p4=1 or p4=2 or p4=3;(1.2+.3(40(p4-1)}p7)/40)p11}p13;jmp 3
21: if p4<p11 or p4>2p11;5300+p1)Z;gto "err stp"
22: cll 'fset'(p1,p4);0}p7;if p4>1.33p11;40}p7;if p4>1.66p11;80}p7
23: if p13-p2/2<p11 or p13+p2/2>2p11;5100+p1)Z;gto "err stp"
24: p2-(p2*X[5])p8)}p9;7}p13)p14;if p6=0;p2}p8
25: if p8>(val(Y$[p5,1+p7,5+p7])*1e6}p11);p2-(p11}p8)}p9;0}p13;1}p15
26: if p9>(val(X$[p5,1+p7,5+p7])*1e6}p11);p2-(p11}p9)}p8;0}p14;1}p16
27: if p15 and p16;5400+p1)Z;gto "err stp"
28: if p15;jmp 3
29: for X=1 to 7;if val(Y$[p5,5X+1+p7,5X+5+p7])*1e6<p8;X-1)p13;10)X
30: next X
31: if p16 or p6=0;jmp 3
32: for X=1 to 7;if val(X$[p5,5X+1+p7,5X+5+p7])*1e6<p9;X-1)p14;10)X
33: next X
34: "fnout":sfq 14;band(Z[p10],65024)}Z[p10]
35: ior(Z[p10],ior(ior(shf(p13,-6),shf(p14,-3)),p6))}Z[p10]
36: wrt 723,"OS",p10,Z[p10],"T":cll '?'(4,1);cfq 14;ret
37: "pulse":if p1<1 or p1>12 or frc(p1);3e3)Z;gto "err stp"
38: if p2<0 or p2>7 or frc(p2);3100+p1)Z;gto "err stp"
39: sfq 14;ior(shf(p2,-9),band(Z[int(p1/2+.5)+4}p3],61951))}Z[p3]
40: wrt 723,"OS",p3,Z[p3],"T":cll '?'(4,1);cfq 14;ret
41: "biph":if p1<1 or p1>12 or frc(p1);4e3)Z;gto "err stp"
42: if p2<0 or p2>7 or frc(p2);4100+p1)Z;gto "err stp"
43: sfq 14;ior(shf(p2,-12),band(Z[int(p1/2+.5)+4}p3],36863))}Z[p3]
44: wrt 723,"OS",p3,Z[p3],"T":cll '?'(4,1);cfq 14;ret
45: "auxmod":if p1<0 or p1>12 or frc(p1);6e3)Z;gto "err stp"
46: if p2<0 or p2>3 or frc(p2);6100+p1)Z;gto "err stp"
47: sfq 14;ior(band(Z[12],1023),ior(shf(int(p1/2+.5),-10),shf(p2,-13)))}Z[12]
48: wrt 723,"OS,12",Z[12],"T":cll '?'(4,1);wait 1
49: wrt 723,"OS,12",band(Z[12],58367)}Z[12],"T":cll '?'(4,1);cfq 14;ret
50: "AMaux":if p1<0 or p1>12 or frc(p1);17000)Z;gto "err stp"
51: if p2<0 or p2>3 or frc(p2);17100+p1)Z;gto "err stp"
52: sfq 14;ior(band(Z[12],65472),ior(int(p1/2+.5),shf(p2,-3)))}Z[12]
53: wrt 723,"OS,12",Z[12],"T":cll '?'(4,1);wait 1
54: wrt 723,"OS,12",band(Z[12],65528)}Z[12],"T":cll '?'(4,1);cfq 14;ret
55: "ampset":if p1<1 or p1>12 or frc(p1);7e3)Z;gto "err stp"
56: if (p2}p4)<0 or p4>81;7100+p1)Z;gto "err stp"
57: if p4=81;70}p4;15}p3;jmp 3
58: if p4=80;70}p4;14}p3;jmp 2
59: p4mod10)p3)p5;if p3>=8;12)p3;if p5=9;13)p3
60: sfq 14;band(Z[11],64512)}Z[11]
61: ior(int(p1/2+.5)-1,ior(shf((p4-p4mod10)/10,-7),shf(p3,-3)))}Z[11]
```

```
 62: wrt 723,"OS,11",Z[11],"T";cll '?'(4,1);cfg 14;ret
 63: "AMown";if p1<1 or p1>12 or frc(p1);8e3)Z;gto "err stp"
 64: if (int(1e3/p2)-X[4])p4)<0 or p4>32767;8100+p1)Z;gto "err stp"
 65: 79+int(p1/2+.5))p5;cll 'AMaux'(p1,2);for X=1 to p3;cll 'AMval'(p1)
 66: if U<0 or U>10X[11];8200+p1)Z;gto "err stp"
 67: wrt 723,"OS",p5,U,"T";wait p4;cll '?'(4,1);next X
 68: wrt 723,"OS",p5,Z[p5-85],"T";cll '?'(4,1);ret
 69: "setVCO";sfg 14;if p1<0 or p1>12 or frc(p1);1e4)Z;gto "err stp"
 70: ior(band(Z[int(p1/2+.5)+4)p2],32767),shf(p1mod2,-15))}Z[p2]
 71: wrt 723,"OS",p2,Z[p2],"T";cll '?'(4,1);cfg 14;ret
 72: "initial";clr 7;trk 1;ldf 91,U$;prt U$;wait 3500;wrt 723,"WF,13.2,1,T"
 73: wait 350;ina Z;for X=2 to 12;wrt 723,"SF",X,2,1,1,"T";wait 100
 74: next X;for X=1 to 6;wrt 723,"OS",X+4,32760)Z[X+4],"T";cll '?'(4,1)
 75: wrt 723,"OB",X+4,15,1,"T,WA,.5T,OB",X+4,15,0,"T"
 76: wrt 723,"OS,12",ior(shf(X,-10),X),"T";cll '?'(4,1)
 77: wrt 723,"OS,11",ior(1008,X-1),"T";cll '?'(4,1);next X
 78: wrt 723,"OS,12,0,T";ret
 79: "stepmod";if p1<1 or p1>12 or frc(p1);11000)Z;gto "err stp"
 80: 2+int((p1-.5)/4))p4
 81: Z[p4])p6;-8)p5;255)p3;if int(p1mod4/2+.5)=1;0)p5;65280)p3
 82: sfg 14;for X=1 to p2;cll 'stepval'(p1)
 83: if U<0 or U>255 or frc(U);11100+p1)Z;gto "err stp"
 84: wrt 723,"OS",p4,ior(shf(U,p5),band(p6,p3))}p6,"T"
 85: cll 'stepwt'(p1);int(U-X[2])}U;if U<0 or U>32767;11200+p1)Z;gto "err stp"
 86: wait U;cll '?'(4,1);next X
 87: wrt 723,"OS",p4,Z[p4],"T";cll '?'(4,1);cfg 14;ret
 88: "ownswp";if p1<1 or p1>12 or frc(p1);9e3)Z;gto "err stp"
 89: 2+int((p1-.5)/4)}p10;-8)p14;255)p12;if int(p1mod4/2+.5)=1;0)p14;65280)p12
 90: band(p12,Z[p10])}p15;cll 'fval#'(p1,p2);W)p7
 91: cll 'fval#'(p1,p3);if (W)p8)<p7;p8)p11;p7)p8;p11)p7
 92: if ((1e3(p3-p2)/p4)p11)/(p8-p7)p6)-X[3])p13)>32767;9100)Z;gto "err stp"
 93: 1)p5;if p13>=0;jmp 2
 94: 0)p13;int(p6*X[3]/p11))p9;if p9<1 or p9>p6;9200+p1)Z;gto "err stp"
 95: for Y=2 to p5+1;for X=p7 to p8 by p9
 96: wrt 723,"OS",p10,ior(shf(X,p14),p15),"T";wait p13;cll '?'(4,1);next X
 97: for X=p8 to p7 by -p9;wrt 723,"OS",p10,ior(shf(X,p14),p15),"T"
 98: wait p13;cll '?'(4,1);next X;next Y;wrt 723,"OS",p10,Z[p10],"T"
 99: cll '?'(4,1);cfg 14;ret
100: "swp175";if p1<1 or p1>12 or frc(p1);14000)Z;gto "err stp"
101: int(p1/2+.5))p6;if p4>1.95e4 or p4<0;14100+p1)Z;gto "err stp"
102: if (p5<0 or p5>11) and (p5<14 or p5>21);14200+p1)Z;gto "err stp"
103: 1+2int((p1-1)/6)-p1mod2)p7;7)p12
104: (2.5e8)2^p7)p10;0)p11;if p2>1.33p10;40)p11;if p2>1.66p10;80)p11
105: if p2+p3/2>2p10 or p2-p3/2<p10;14300+p1)Z;gto "err stp"
106: for X=1 to 7;if (val(W$[p6,5X+1+p11,5X+5+p11])}p14)>=p3/1e6;X)p12;10)X
107: next X;if X<9;val(W$[p6,5p12+1+p11,5p12+5+p11])}p14
108: val(W$[p6,5p12-4+p11,5p12+p11])}p13
109: (X[13]/(p14-p13)}p13)(p3/1e6)+(X[12]+p12*X[13])-p13p14)p8
110: if p8<1e-3 or p8>X[10];14400+p1)Z;gto "err stp"
111: ""}U$;flt 1;"F"&str(p4)}U$;"E"}U$[6,6];fxd 2
112: U$&"A"&str(p8)&"C"&str(p5)&"DOP1I"}U$
113: cll 'fset'(p1,p2);wrt 701,U$;cll 'auxmod'(p1,3);ret
114: "AM175";if p1<1 or p1>12 or frc(p1);1.8e4)Z;gto "err stp"
115: if p3<0 or p3>1.95e4;18100+p1)Z;gto "err stp"
116: if (p4<0 or p4>11) and (p4<14 or p4>21);18200+p1)Z;gto "err stp"
117: if (p2/X[11])p5)<1e-3 or p5>X[14];18300+p1)Z;gto "err stp"
118: ""}U$;flt 1;"F"&str(p3)}U$;"E"}U$[6,6];fxd 2
119: U$&"A"&str(p5)&"D"&str(p5/2)}U$;fxd 0;U$&"C"&str(p4)&"P1I"}U$
120: wrt 702,U$;cll 'AMaux'(p1,3);ret
121: "DC175";for X=4 to p0;if pX<1 or pX>12 or frc(pX);15000+X)Z;gto "err stp"
122: int(pX/2+.5))p11;if p(11+p11);15100)Z;gto "err stp"
```

```
123: 1)p(11+p11);next X;if p3#1 and p3#2;15200)Z;gto "err stp"
124: if p1<0 or p1>100;15300)Z;gto "err stp"
125: if p2<5.1e-2;15400)Z;gto "err stp"
126: 0)p13;255)p12;if p1<=50;255-int(2.54(50-p1)))p12;jmp 2
127: int(2.54(p1-50)))p13
128: fxd 0;""}U$;"W"&str(p12)&"V"&str(p13)&"U1A2D1C2P0"}U$
129: flt 1;U$&"F"&str(1e3/p2)&"I"}U$;cap(U$)}U$;jmp p3
130: wrt 702,U$;jmp 2
131: wrt 701,U$
132: for X=4 to p0;cll 'pulse'(pX,p3+2)
133: next X;ret
134: "T/P";if p1<2e-3 or p1>1e7;16000)Z;gto "err stp"
135: wrt 723,"OS,13",p1/2,"T";cll '?'(4,1);p1)Z[13]
136: for X=2 to p0;if pX<1 or pX>12 or frc(pX);16100+X)Z;gto "err stp"
137: int(pX/2+.5))p9;if p(9+p9);16200)Z;gto "err stp"
138: 1)p(9+p9);cll 'pulse'(pX,5);next X;ret
139: "special";if p1>3 or p1<0 or frc(p1);12000)Z;gto "err stp"
140: ((8)p6)+1)p7)+1)p8;4)p5;if p1<2;2)p5;((5)p6)+1)p7)+1)p9
141: if p4<0;12100+p6)Z;gto "err stp"
142: int(1e3/p2-X[1]))p6;if p6<0 or p6>32767;12200)Z;gto "err stp"
143: int(p2p3))p7;for X=0 to 2p7 by 2;cll 'valspec'(p4)
144: wrt 723,"OP" ;5,U,3,W,p6,Y,p7,Z,p8,V,"T";wait p6;cll '?'(2,1);next X;ret
145: "err stp";fxd 0;wrt 701,"POI";wrt 702,"POI"
146: wrt 723,"OP,5,32760,6,32760,7,32760,8,32760,9,32760,10,32760,T"
147: (int(Z/1e2))p24)-10(int(Z/1e3))p21))p22;Z-100p24)p23
148: fmt 1,"fault: ",f2.0,2x,f1.0,x,f2.0,2/;wrt 16.1,p21,p22,p23
149: dsp "* error ",Z," *";beep;wait 1250;dsp "";wait 99;jmp 0
150: "shutoff";prt "stopped / error";wrt 701,"POI";wrt 702,"POI";fxd 0
151: wrt 723,"OP,5,32760,6,32760,7,32760,8,32760,9,32760,10,32760,T"
152: fmt 1,"error ",c,f3.0,/,"line#",f3.0,2/;wrt 16.1,char(rom),ern,erl
153: dsp "* shutdown *";beep;wait 1250;dsp "";wait 99;jmp 0
154: "enter";1)p1;32)p2;if pos(cap(U$),"G"))p3;p3-1)p2;1e9)p1;gto "}"
155: if pos(cap(U$),"K"))p3;p3-1)p2;1e3)p1;gto "}"
156: if pos(cap(U$),"U"))p3;p3-1)p2;1e-6)p1;gto "}"
157: if pos(U$,"M"))p3;p3-1)p2;1e6)p1;gto "}"
158: if pos(U$,"m"))p3;p3-1)p2;1e-6)p1;gto "}"
159: if pos(cap(U$),"D"))p3;p3-1)p2;1)p1;gto "}"
160: if pos(cap(U$),"H"))p3;p3-1)p2;1)p1;gto "}"
161: if pos(cap(U$),"S"))p3;p3-1)p2;1)p1;gto "}"
162: if pos(U$,"e");32)p2;1)p1;gto "}"
163: "}";ret val(U$[1,p2])*p1
164: "inRFid";"6b"}U$;ent "RF#(1-6), VCO-(A,B)",U$[1,2];cap(U$)}U$
165: if (num(U$[1,1]))p1)>48 and p1<55;jmp 2
166: dsp "bad form; reenter (e.g.; '6a')";wait 2500;jmp -2
167: if (num(U$[2,2]))p1)#65 and p1#66;jmp -1
168: ret 2val(U$[1,1])+p1-66
169: "loadY$":trk 1;for X=1 to p0
170: if pX<1 or pX>12 or frc(pX);13000)Z;gto "err stp"
171: if p((int(pX/2+.5))p13)+6);13100+pX)Z;gto "err stp"
172: p13)p(p13+6);ldf pX+6,V$;V$)Y$[p13];next X;ret
173: "loadX$":trk 1;for X=1 to 2p0-1 by 2
174: if pX<1 or pX>12 or frc(pX);13200)Z;gto "err stp"
175: if (p(X+1))p20)<1 or p20>5 or frc(p20);13300+pX)Z;gto "err stp"
176: if p((int(pX/2+.5))p19)+12);13400+pX)Z;gto "err stp"
177: p19)p(p19+12);ldf 5pX+p20+13,V$;V$)X$[p19];next X;ret
178: "loadW$":trk 1;for X=1 to p0
179: if pX<1 or pX>12 or frc(pX);13500)Z;gto "err stp"
180: if p((int(pX/2+.5))p13)+6);13600+pX)Z;gto "err stp"
181: p13)p(p13+6);ldf pX+78,V$;V$)W$[p13];next X;ret
```

REFERENCES

[1] H. M. Kaye, *Multimode Guidance Project Low Frequency ECM Simulator: Hardware Description,* JHU/APL TG 1335A (Oct 1982).

[2] J. M. Van Parys, *MMG ECM Simulator Software, Interim Documentation,* JHU/APL F4D-4-80(U)-002 (30 Sep 1980).

BIBLIOGRAPHY

1. Hewlett-Packard, Desktop Computer Division, *9825 Desktop Computer Operator's Guide,* part no. 09825-90000, Ft. Collins, Colo. (1979).
2. Hewlett-Packard, Desktop Computer Division, *String ROM Programming,* part no. 09825-90020, Ft. Collins, Colo. (1979).
3. Hewlett-Packard, Desktop Computer Division, *Advanced Programming ROM,* part no. 09825-90021, Ft. Collins, Colo. (1979).
4. Hewlett-Packard, Desktop Computer Division, *Matrix ROM Programming,* part no. 09825-90022, Ft. Collins, Colo. (1979).
5. Hewlett-Packard, Desktop Computer Division, *General I/O ROM Programming,* part no. 09825-90024, Ft. Collins, Colo. (1979).
6. Hewlett-Packard, Desktop Computer Division, *Extended I/O ROM Programming,* part no. 09825-90025, Ft. Collins, Colo. (1979).
7. Hewlett-Packard, Desktop Computer Division, *6942A Multiprogrammer User's Guide,* part no. 06942-90003, Ft. Collins, Colo. (1979).
8. Hewlett-Packard, Desktop Computer Division, *9825A Software General Utility Routines,* part no. 09825-10001, Rev. G., Ft. Collins, Colo. (no date).
9. Wavetek, *Instruction Manual, Model 175 Arbitrary Waveform Generator,* San Diego, Calif. (May 1980).

INITIAL DISTRIBUTION EXTERNAL TO THE APPLIED PHYSICS LABORATORY*

The work reported in TG 1335B was done under Navy Contract N00024-83-C-5301. This work is related to Task A3G0, which is supported by Naval Sea Systems Command (SEA-62R55).

ORGANIZATION	LOCATION	ATTENTION	NO. OF COPIES
DEPARTMENT OF DEFENSE			
DTIC	Alexandria, VA		12
Department of the Navy			
NAVPRO	Laurel, MD		1
OASN (RE&S)	Washington, DC	R. Rumpf	1
CNO	Washington, DC	OP-352L	1
NAVSEASYSCOM	Washington, DC	SEA-00	1
		SEA-62	1
		SEA-62Q	1
		SEA-62R	1
		SEA-62R1	1
		SEA-62R5	1
		SEA-62R55	5
		SEA-62R57	1
		SEA-62Z3	1
		SEA-62Z31F	1
		SEA-9961	2
		PMS-400B	1
		PMS-400M	1
NAVAIRSYSCOM	Washington, DC	AIR-03P2	1
		AIR-360	1
		AIR-360E	1
		AIR-50174	2
Naval Surface Weapons Ctr.	Dahlgren, VA	F-14	2
		F-42	1
		F-46	1
		Library	1
Naval Weapons Ctr.	China Lake, CA	3906	5
		3921	1
		39501	1
		Library	1
Pacific Missile Test Ctr.	Pt. Mugu, CA	1241	2
Department of the Army			
MICOM	Huntsville, AL	DRSMI-ROA	2
Department of the Air Force			
Aeronautical Systems Div.	Dayton, OH	YYM	1
CONTRACTORS			
Hughes Aircraft Co./MSD	Canoga Park, CA	L. J. Hassencahl	2
General Dynamics/Pomona Div.	Pomona, CA	H. J. Meltzer	2

Requests for copies of this report from DoD activities and contractors should be directed to DTIC, Cameron Station, Alexandria, Virginia 22314 using DTIC Form 1 and, if necessary, DTIC Form 55.

*Initial distribution of this document within the Applied Physics Laboratory has been made in accordance with a list on file in the APL Technical Publications Group.

END

DATE
FILMED

4-83

DTIC