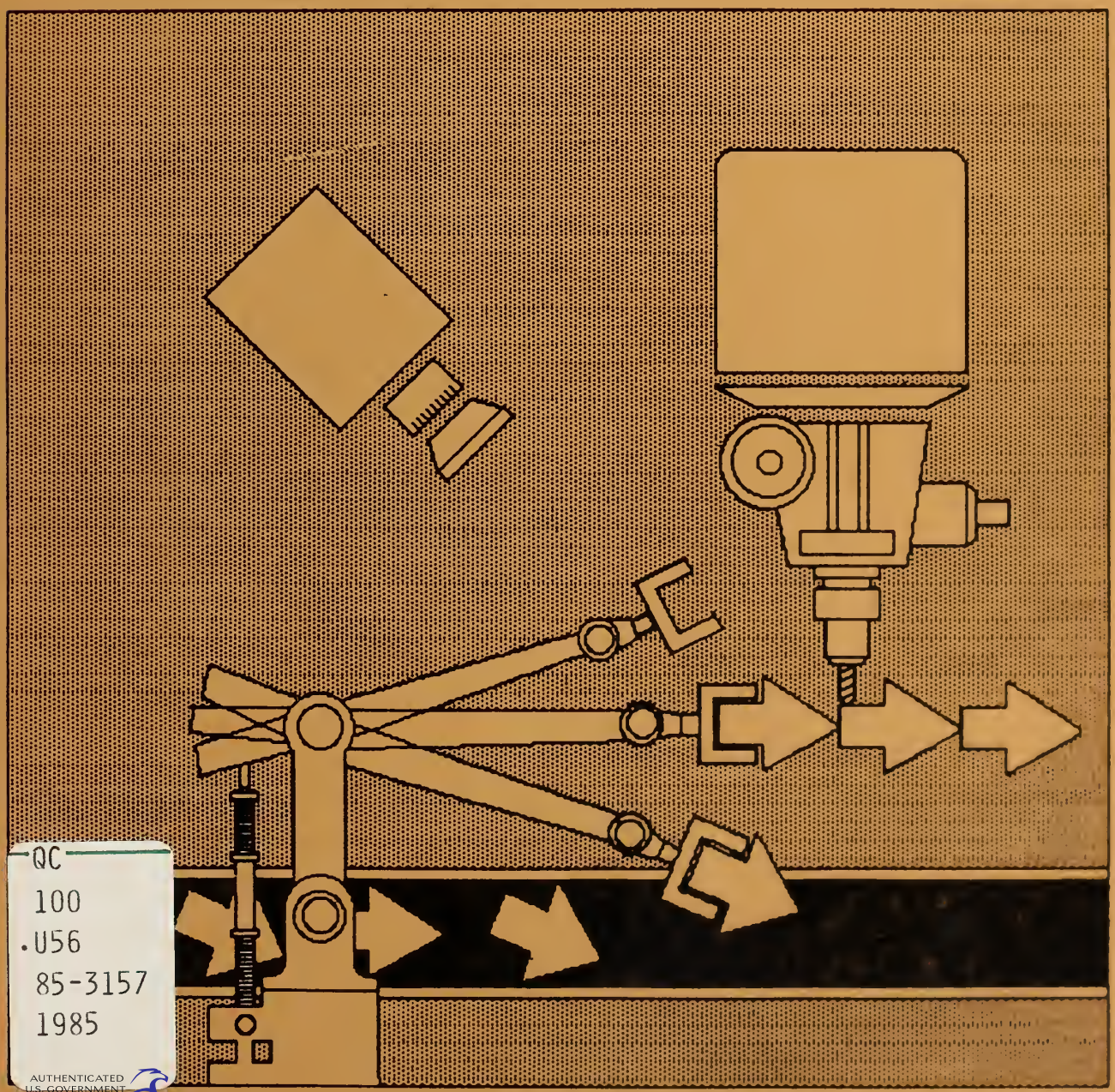
U. S. Department of Commerce

National Bureau of Standards

NBS-IR-85-3157

Hierarchical Control System Emulation Programmer's Manual

January 1985



NBS-IR-85-3157

Hierarchical Control System Emulation Programmer's Manual

Cita M. Furlani, Editor

January 1985

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Factory Automation Systems Division
Gaithersburg, MD 20899

Acknowledgement

The major part of the work that is represented in this manual was done by Bolt Beranek and Newman, Inc. under Department of Commerce contract NB81SBCA0826 entitled "Emulation/Simulation of an Automatic Manufacturing Test Facility". This manual in its original form was derived from the final report (dated October 1982) from that contract. The product has been further modified at the National Bureau of Standards. This manual documents the state of the Hierarchical Control Systems Emulator as of January 1985.

TABLE OF CONTENTS

1. HCSE PROGRAMMER'S MANUAL

This manual is intended to provide a guide to the prospective maintainer/modifier of the HCSE software. The reader is assumed to be familiar with operation of the emulation software (User's Manual) as well as the following: Praxis language, Fortran, and VMS system calls. The code can be broken into three relatively independent classes. The first, parsing, reads FSM source files and constructs Praxis code or dictionaries. Also in this class is the builder module which writes Praxis code to invoke various state machines within a single VMS process. The second class of modules creates the run time environment; this includes the shared memory and display modules. The third class of modules does post processing of emulation log files to produce listings and summaries.

2. HARDWARE AND SOFTWARE REQUIREMENTS

Hardware

DEC* VAX computer system
DEC supported display terminal (VT100, VT52, etc)

Software

DEC VAX/VMS operating system
Praxis compiler and text I/O
DEC Fortran 77

---

* Certain commercial products are identified in this manual in order to adequately describe the HCSE. Such identification does not imply recommendation or endorsement by the National Bureau of Standards.

3. EMULATION SOFTWARE OVERVIEW

3.1 Philosophy of Emulation

The HCSE emulation package is designed to provide a flexible framework for the development of finite state control systems. There were several general design goals which drove the development into its present implementation.

1) The system must represent all manner of state machines including those with "extended" interpretation.

2) The system had to have the potential of running in real time.

3) The system had to be both observable and produce an audit trail.

The first goal resulted in the choice of a compiled rather than interpretive approach. While low level control modules are relatively easy to represent as pure state machines, higher level functions often require a more expressive vehicle. Praxis presented a completely general, modern, strongly typed language, which could be utilized to expand the state machine concept wherever it was essential. However, the goal of representing each module, at least in the macro sense, as a state machine, required developing a pseudo language which could be translated to Praxis.

Goal number two drove the implementation to a multiprocess,

shared global section implementation which promoted use of many VMS features. Most of these features would have been duplicated in any approach which ran the simulation as a single process. VMS provides detailed, per process, audit information which allows complete partitioning of system resources. Also, the emulation itself maintains a log of all user-defined type declarations and variable transitions stored in common memory.

3.2 Summary Software Description

This section describes in words the function and interrelationships between the various software files. It is broken into three sections to reflect the general classes of software; however the distinctions are in some cases open to question.

3.2.1 Module parsing and building

State machine modules are all parsed by a single subroutine regardless of the intended use of the parsed product. Modules which require a parsed product are the state machine translator, PARSER, and the data dictionary, DICTION. They both request parsed products from module PARSING.

3.2.1.1 PARSING.PRX

The parsing module reads an 'FSM' file to construct an in-memory, parsed representation of the file. It collects lists of types (predefined and user-defined) and variables (input, output, state, and internal). When type and variable declarations are read, the information obtained from the FSM module is further dissected into more detailed bits of data to be digested by the parser. For example, the following code

```
//name   WS1
//include GLOBAL.DAT
//type mailbox structure   | network communications mailbox
         mb_length : integer_2   | length of mailbox in bytes
         mb_seqno : integer_2
         mb_time : integer_4
         mb_text : string
  endstructure
//input command mailbox   | command from job cell
```

is broken down into two individual sets of data. Type information is dissected first into the following code.

type_name	MAILBOX
type_class	STRUCTURE
type_fsm	WS1
type_comm	network communications mailbox

Type information is broken down further by class specifications.

str_nelms	4 (number of structure elements)
str_elm	a pointer which indicates the storage location of the structure's elements in common memory.

The two fields listed above are only inherent to the STRUCTURE type-class. Types of the ARRAY class are dissected into two fields named arr_nelms (number of elements in array) and arr_elm_type (type of the elements). ALIAS types are dissected into alias_name (name of alias type) and init_value (value at which alias type is initialized). The final type-class recognized by the parser is BASIC which includes 14 predefined types that contain only one field. (Note: The only interaction that a HCSE programmer may have with BASIC types is declaring user-defined types in terms of them.) BASIC types are discussed here simply to provide the reader with a better understanding of how the parser handles type definitions.

In the case of STRUCTURE type-classes, even further dissection is required. Elemental information needs to be analyzed. The following fields represent the first element of the structure MAILBOX.

elm_name	MB_LENGTH
elm_type	INTEGER_2
elm_comm	length of mailbox in bytes

The remaining elements in the structure are broken down in the same manner. The dissection of elements into this format takes place only when the STRUCTURE type_class is specified.

When type dissection is complete, the information attained will be added to the list of other type declarations including those found in the files STANDARD.ISM and GLOBAL.DAT (The use of type declaration files is described in the HCSE User's Manual). To recap briefly, STANDARD.ISM is a file which contains common type declarations and is called automatically by the parser. GLOBAL.DAT is also a type declaration file written by the HCSE user and may contain type declarations (formatted in the same manner as types are declared in FSM modules using the //type statement) that suit a particular process or emulation. This included file (//include) must appear in the same directory where the FSM modules are parsed or in directories with the assigned logical names FSM$INCLUDE_0 through FSM$INCLUDE_4. The //INCLUDE statement is used to avoid declaring the same type definitions in separate FSM modules. The included file is attached to the list of type declarations of an FSM module at the precise point where the //include statement appears in the FSM. The other set of declarations handled by the parser are variable declarations. Continuing with the FSM example code given above, the variable command would be broken down as follows.

```
var_name        COMMAND
var_type        MAILBOX
var_comm        command from job cell
var_fsm         WS1
```

This information is added to either the list of variables read from common memory (//input, //inparameter, and //state), written to common memory (//state, //output, and //outparameter), or kept internal (//internal).

The actual state transition table is also constructed in memory by PARSING.PRX. This table is constructed with a list of lists. The top level list consists of one entry for each condition/action pair. Stored in the entry are two pointers, one to a list of conditions, the other to a list of actions. Each condition or action is one entry on these lists. Note that each condition or action line may contain multiple conditions or actions separated by semicolons, resulting in several entries on this list. For example, the pair of lines

```
//conditions curs = "start"; sensor = 3
//actions nexts := "running"; actuator := 4.3
```

is stored as one entry on the primary list (rowlist), with pointers to two lists (conditions, actions) each of which will contain two entries. These entries will look like

```
conditions  col_text:        TST(curs,"start")
            col_text:        sensor = 3
actions     col_text:        nexts := "running"
            col_text:        actuator := 4.3
```

There are several special cases which are handled identically to actions. These are //preprocess, //postprocess,

//nomatch, and //multimatch. In each case, an action list is constructed.

The parsing module terminates as soon as it encounters a //procedures statement, or end of file. The input file is left open, and pointing to the next record after the procedures statement.

3.2.1.2 PARSER.PRX

Parser, or more properly, translator, converts the internal format generated by parsing to true Praxis code. It declares each module as a function, named the FSM name, which returns an integer compute_delay. The Praxis code imports the various common memory and textio routines. User-defined types are declared in the precise form that appeared in the FSM source file and are merely copied over to the Praxis file. Variables are declared (static) and a dummy constant with n_variable name is constructed to hold the name string. The remainder of the input file is copied to the Praxis file. Code is added which collects variables from common memory and does common memory synchronization. The preprocessing code is entered. The state transition table is constructed in two sections. The first section consists of one "if" statement for each condition/action pair. If the "if" triggers, the pair number is saved. Multiple triggers are detected by the pair number already having a value when an "if"

triggers. If this happens, the pair number is set to -1, signaling a multimatch condition. The second portion of the state table consists of the actions. This is constructed from a giant "select" on pairnumber, with pair number -1 reserved for multimatches and zero for no matches. After this table, any post processing code is entered. Finally, output variables are stored for writing to common memory.

As a parting shot, the internal variable "first_entry" is set false. "First_entry", which is initialized "true" the first time a module is entered, provides a simple mechanism for state machines to perform graceful startups, regardless of their initial state.

3.2.1.3 BUILDER.PRX

The BUILD module generates PRAXIS code and a linker options file which are used to construct VMS images. The builder takes commands of the form

```
$ BUILD image FSM1/10, FSM2, FSM3/20
```

where image is the image file to be constructed, and FSM1, etc. are state machine modules. The option on each module is the scheduling interval in "ticks" with the default (zero) being every tick. The builder constructs two PRAXIS variables for each

module; FSM1_count and FSM1_delay. These variables are used to store the current scheduling interval of the modules. FSM_count is decremented whenever it is non-zero and the module is activated whenever the count reaches zero. On each "tick", the builder places one call to CM_DUMP_OUTPUTS (in SHAREOUT.PRX) to write all variables from all modules which were activated.

3.2.1.4 DICTION.PRX

The dictionary program may be used to output a list of user-defined type declarations and construct a cross reference listing of module inputs and outputs. It uses PARSING to collect the information.

3.2.2 Run Time Routines

The run-time environment consists of the common memory routines and the run-time display.

3.2.2.1 SHRMEM.FOR

This module, written in Fortran 77, implements reading and writing to common memory, read/write synchronization, type and variable logging, and statistics gathering.

3.2.2.2 SHAREOUT.PRX

This module acts as a buffer between the state machines and the actual writes to common memory. It implements the output delay function, and also the feature which allows multiple state machines to exist independently of each other in a process.

3.2.2.3 DISPLAY.PRX

This is the run-time display. It displays a list of variables which the user has requested. On each "tick", DISPLAY reads its list of variables from common memory, just as any other emulation process (it is not a "privileged" process), and displays the values of all variables which have been requested. It also provides timing routines which can force the emulation to run at some multiple of real time. Again, this is done without special "privilege". Basically, the simulation will not advance until all processes assent to it; DISPLAY simply delays before assenting. The DISPLAY program also provides access to the emulation snap shot facility.

3.2.3 Post Processing

Post processing is built entirely on the log files provided by the emulation. These files contain a list of the user-defined type declarations, all variable transitions, and some general emulation and CPU statistics. Each record in the log file begins with a single character identifier which classifies the records

according to the information they contain. The following is a list of these identifiers.

T - Type information (user-defined) : contains the type name, class, and size. If the type is ALIAS, the record will also contain the type handle; if ARRAY, record contains handle and number of type elements; and if STRUCTURE, record contain the number of components.

C - Component information : contains the name and type handle of the components of a STRUCTURE type.

V - Variable information : contains variable name, logging time in ticks, type handle, and variable value. 32 bytes of storage is allocated to hold the variable value.

D - Data : serves as a continuation line for a variable record. If the required storage for a variable value exceeds 32 bytes, the initial portion of this value is placed in the variable record, and the remaining portion will be retained in successive data records.

S - Statistics : contains general information such as the image name, tick_spacing, and CPU and timing statistics.

3.2.3.1 SIMLIST.PRX

SIMLIST reads through the emulation log file one record at a time and transforms the logged information into user readable form. The final listing contains some opening statistics, user-

defined type descriptions, chronological records of variable transitions, and finally, some closing statistics. SIMLIST does no interpretation of the data it reads, it just reformats.

3.2.3.2 SUMMARY.PRX

SUMMARY reads information from log files to accumulate statistics on individual variables. This program derives the minimum, maximum, time that each variable spent in a given state, and the mean and variance for real valued variables.

3.2.4 Libraries

Libraries fall into two classes: those which are real, i.e., have object code, and those which have Praxis versions just to generate synopsis files.

3.2.4.1 VAXRUNTIM.PRX

This file contains the Praxis description of all run-time routines which can be called without the calling program having to specifically link with them. The current version is by no means complete. It represents the set of VMS system services and routines from the VAX run-time library that have been required to date.

3.2.4.2 VAXDEF.PRX

VAXDEF has several small functions which implement some of the VMS calling conventions, such as passing strings by description. This module has real object code.

Documentation from file : PARSING.PRX

name: PARSING

func: this module parses FSM files for all programs in the emulation package which require it. The module contains all the code which is dependent on the particular syntax of the state machine language. The outputs of this module consist of dynamic lists of parsed source code. The module also exports to other modules the internal structure of these lists. The exported information consists of: string sizes and defaults for variable names, types, and line lengths; a structure which contains all type names, the modules which declare them, and further type information which is dependent upon the class of the type; a structure which contains all the information on declared simulation variables; lists of variables as input, output, state, or internal; a structure which holds pointers to condition/action pairs; a list of condition action pairs, which points to lists of code for each column; lists of actions for preprocessing, postprocessing, multimatch, and nomatch.

call: call COLLECT_FSM(ttyfile, infile, procedures)

args: ttyfile - inout file : a textio file used for errors and messages infile.

file - inout file : a textio file with the input file open on it.

procedures - out boolean : set true if the file had procedure code left in it.

Documentation from file : PARSER.PRX

name: PARSER

func: this program calls the PARSING module to convert state machine language into internal format. It then converts this internal format into Praxis code. The code which this module generates is responsible for all common memory synchronization and access. Code is generated to declare all user-defined types, to read all input variables from common memory at the beginning of each cycle, do any preprocessing, determine which condition/ action pair to execute, execute it, do postprocessing, and store outputs in the queue waiting for common memory write access.

call main program runs as a VMS foreign command, i.e., $ PARSE WS1.

args: input file, extension .FSM assumed and required.

refs: COLLECT_FSM (in module PARSING)
Textio
LIB$GET_FOREIGN

Documentation from file : BUILDER.PRX

name: BUILDER

func: BUILDER is a main program which constructs the Praxis code to call one or more state machines, as well as generating the linker options file to specify the object modules necessary to create the executable image. Builder is run as a VMS foreign command. It accepts commands of the following format:

$BUILD EXENAME FSM1, FSM2/10, FSM3, FSM4/50, etc.

where BUILD has been defined as a VMS foreign command, EXENAME is the name of the executable image to be built, FSM1 ... FSMn are state machine

names. Each state machine name may be optionally qualified by an integer. This cycles between invocations of this state machine, i.e., FSM2/2 means call state machine named FSM2 every third common memory cycle. The default, zero, causes machines to be activated for every common memory cycle. It is important to realize that activation of a FSM requires reading and writing ALL its input/output variables; this represents a substantial fraction of total CPU cycles in a emulation run; therefore, it is advisable to set these numbers as high as is reasonable.

call: VMS foreign command: $ BUILD CONTROL WS1/10, WS2/10, WS3, etc.

args: name of main module which will contain specified state machines; a list of state machines to be executed as part of this process.

refs: Textio
LIB$GET_FOREIGN

Documentation from file : DICTION.PRX

name: DICTION

func: dictionary module. This routine accepts a list of state machine modules (as a foreign command), parses them, and forms two lists: a list of user declared types listing declarers and comments, and a list of variables listing readers, writers, and comments.

Format: DICTIONARY is defined as the foreign command

```
$ DICT*IONARY :== $HCSE_LIBRARY:DICTION
```

Command syntax:

```
$ DICT JC1, WS1, WS2
```

Output goes to the logical unit DICT_OUTPUT if the logical name exists and can be opened. Otherwise,

it goes to SYS$OUTPUT.

call: main module, run as VMS foreign command, i.e., $ DICT JC1, WS1, WS2, etc.

args: a list of source files for state machine modules. The file name only should be provided; the file type will be ".FSM". If a file name starts with an "@", it is assumed to be the name of a file containing FSM source file names; the file type ".DAT" (rather than ".FSM") is appended and further input is taken from the file. This indirection can be nested up to the capabilities of textio.

refs: COLLECT_FSM (in module PARSING)
Textio
LIB$GET_FOREIGN
LIB$DATE_TIME

Documentation from file : SHRMEM.FOR

Common memory is implemented as a Fortran common area which is dynamically mapped by each process in the simulation. This mapping is accomplished using the VMS mapped global section primitives. Types and variables are stored in common memory by ASCII name, up to 32 bytes long. Associated with types and variables are the following:

- a pointer into a local heap storage (in the mapped section)
- the time the variable was last written
- a flag controlling whether variable transitions will be logged
- a pointer into arrays of type information
- a list of FSMs that read this variable

Variables and user-defined types are allocated sequentially and are added to common memory on the first reference to them. On the first reference, the variable or type is initialized using a file that is keyed on variable or type name. The file has the logical name 'AMRF'. This only occurs if the file exists and the variable or type has a value recorded in it. Access to reading, writing, and linked lists is controlled using cluster event flags, which are assigned as follows:

* 96 - when set, enables reading from common memory
* 97 - when set, enables writing to common memory
* 98 - used as a semaphore controlling write access to all linked lists
* 99 - when set, global shared area has been or is being created
* 100 - when set, global shared area is available for mapping

Subroutines in SHRMEM.FOR

name: CM_IN_MEMORY_ONLY

func: controls whether common memory changes are updated in the keyed access file.

call: call CM_IN_MEMORY_ONLY(newflag,prevflag)

args: newflag - in fortran boolean : when set to true, keyed access file will be updated.

prevflag - previous state of in_memory flag.

refs: none

name: CM_LOGGING_ON

func: controls whether all variable transitions are logged in the logging file. When logging is not on, only specified variables are logged. Types and statistics are logged in any case.

call: call CM_LOGGING_ON(newflag, prevflag)

args: newflag - in fortran boolean : when set false, process-wide logging is disabled.

prevflag - previous state of logging flag.

refs: none

name: CM_TICK_SPACING

func: real function which returns time between simulated clock ticks in seconds.

call: ts = CM_TICK_SPACING( )

args: none

refs: CM_GET_TIME

name: CM_GET_TIME

func: returns current simulation time in ticks as well as tick spacing. (Note: this routine establishes the tick spacing for the entire simulation. Tick spacing is constant.)

call: call CM_GET_TIME(ticks, tick_spacing)

args: ticks - out integer : number of elapsed ticks since emulation started.

tick_spacing - out real : spacing between ticks in real seconds.

refs: CM_SIMJOIN

name: CM_READ_REQUEST

func: grants access to common memory for reading. This routine MUST be called prior to issuing any CM_READ_VARIABLE calls. Synchronization is performed using group event flags. When flag 96 is set, reading is enabled. This routine also bumps an access count. This count must return to zero before writing may take place.

call: changed = CM_READ_REQUEST(fsm_name, fsm_handle)

args: fsm_name - in byte array(32) : name of FSM requesting access.

fsm_handle - inout integer : handle to FSM, MUST be 0 on initial call.

refs: CM_SIMJOIN
CM_WRITE_DONE
CM_FIND_FSM
SYS$WAITFR

name: CM_READ_DONE

func: performs common memory synchronization. This routine MUST be called after all variables have been read on a given tick. The routine decrements an access counter which MUST be 0 for writing to take place.

call: call CM_READ_DONE( )

args: none

refs: none

name: CM_WAKE_FSM

func: wakes up a given FSM, even if none of its inputs

have changed.

call: call CM_WAKE_FSM(fsm_name, fsm_handle)

args: fsm_name - in byte array(32) : name of FSM requesting access.

fsm_handle - inout integer : handle to FSM; MUST be 0 on initial call.

refs: CM_FIND_FSM

name: CM_DISABLE

func: disables an FSM; it will remain disabled even if its inputs change.

call: call CM_DISABLE(fsm_name, fsm_handle)

args: fsm_name - in byte array(32) : name of FSM requesting access.

fsm_handle - inout integer : handle to FSM, MUST be 0 on initial call.

refs: CM_FIND_FSM

name: CM_ENABLE

func: enables a disabled FSM, though without necessarily waking it up.

call: call CM_ENABLE(fsm_name, fsm_handle)

args: fsm_name - in byte array(32) : name of FSM requesting access.

fsm_handle - inout integer : handle to FSM, MUST be 0 on initial call.

refs: CM_FIND_FSM

name: CM_FIND_FSM

func: returns the handle to a given FSM name.

call: call CM_FIND_FSM

args: fsm_name - in byte array(32) : name of FSM requesting access.

fsm_handle - inout integer : handle to FSM, MUST be 0 on initial call.

refs: CM_SIMJOIN
SYS$CLREF
SYS$WAITFR
SYS$SETEF

name: CM_OPEN_VARIABLE

func: returns a handle to be used with all subsequent calls concerning this variable. All new variables are added to linked lists tables.

call: handle = CM_OPEN_VARIABLE(name, typ_handle, fsm_name, fsm_handle, in_or_out)

args: name - in byte array(32) : containing variable names, NUL filled.

typ_handle - inout integer : handle to variable type.

fsm_name - in byte array(32) : FSM name of calling routine.

fsm_handle - inout integer : handle to FSM, MUST be 0 on initial call.

in_or_out - in byte : character I, O, or S, signifying input, output, or state.

refs: CM_SIMJOIN
CM_FIND_VAR
CM_FIND_FSM
SYS$CLREF
SYS$WAITFR
SYS$SETEF

name: CM_READ_VARIABLE

func: reads a specific variable from common memory.

call: CM_READ_VARIABLE(iptr, time, variable)

args: iptr - in integer : containing variable handle.

time - out integer : time (in ticks) that variable was last written.

variable - out general : contents of the variable in common memory.

refs: none

name: CM_WRITE_REQUEST

func: grants access to common memory for writing. This routine MUST be called prior to issuing WRITE_VARIABLE calls. Synchronization is performed using cluster event flags. When flag 97 is set, writing is enabled. This routine controls the transition from system-wide reading to writing. Transition (clearing flag 96 and setting flag 97) occurs when the number of processes awaiting write privileges is >= the total number of processes which have 'joined' the simulation.

call: call CM_WRITE_REQUEST()

args: none

refs: CM_SIMJOIN
CM_READ_DONE
SYS$CLREF
SYS$SETEF
SYS$WAITFR

name: CM_WRITE_DONE

func: performs common memory synchronization. This routine must be called after all writing has been

done. This routine also causes the transition from writing to reading. This transition occurs when the count of processes requesting write privileges goes to zero. At this time, the writing flag (97) is cleared and the reading flag (96) is set. The simulation clock tick is also incremented at this point.

call: call CM_WRITE_DONE( )

args: none

refs: SYS$CLREF
SYS$SETEF

name: CM_WRITE_VARIABLE

func: writes a specific variable to common memory. If this write causes a transition and logging is turned on, then the variable is logged in the logging file. If common memory is being maintained in the keyed access file, then the record is updated in the keyed file.

call: call CM_WRITE_VARIABLE(iptr, variable)

args: iptr - in integer : containing variable handle.

variable - in general : the variable to be written to common memory.

refs: none

name: CM_LOG_VARIABLE

func: establishes the value of the logging flag for an individual variable. This has effect when process-wide logging has been turned off with CM_LOGGING_ON.

call: call CM_LOG_VARIABLE(iptr, newflag, prevflag)

args: iptr - in integer : containing variable handle.

newflag - in fortran boolean : when set true,

logging is enabled for a specific variable.

prevflag - out fortran boolean : previous state of flag.

refs: none

name: CM_OPEN_ALIAS_TYPE

func: one of a set of type declaration routines. The routine looks up a type in common memory. If the type was not previously declared, it is created. If it had been previously declared, the routine verifies whether the current information on the type corresponds to the previously declared information. Finally, the routine returns a handle to the position of the type information in the type_info arrays. This particular routine works on user-declared types which are declared to be the same as another type (either predefined or user-defined).

call: handle = CM_OPEN_ALIAS_TYPE(name, a_typ)

args: name - in byte array(32) : containing type name, NUL filled.

a_typ - in integer : handle of alias type.

refs: CM_SIMJOIN
SYS$CLREF
SYS$WAITFR
SYS$SETEF

name: CM_OPEN_ARRAY_TYPE

func: one of a set of type declaration routines. This routine performs basically the same operations as the routine CM_OPEN_ALIAS_TYPE described above, except that it is designed to handle user-declared array types as opposed to alias types.

call: handle = CM_OPEN_ARRAY_TYPE(name, elm_type, elm_count, pflag)

args: name - in byte array(32) : containing type name, NUL filled.

elm_type - in integer : pointer to type of the array elements.

elm_count - in integer : number of elements in the array.

pflag - in boolean : flag indicating whether array is packed.

refs: CM_SIMJOIN
SYS$CLREF
SYS$WAITFR
SYS$SETEF

name: CM_OPEN_STRUCTURE_TYPE

func: one of a set of type declaration routines. This routine performs basically the same operations as the routine CM_OPEN_ALIAS_TYPE described above, except that it is designed to handle user-declared structure types as opposed to alias types.

call: handle = CM_OPEN_STRUCTURE_TYPE(name, f_names, f_types, f_count, pflag)

args: name - in byte array(32) : containing type name, NUL filled.

f_name - in byte array(32,*) : list of field names.

f_types - in integer array(*) : list of field types.

f_count - in integer : number of fields.

pflag - in boolean : flag indicating whether structure is packed.

refs: CM_SIMJOIN
SYS$CLREF
SYS$WAITFR

SYS$SETEF

name: CM_FIND_TYPE

func: logical function which returns .true. if the given type currently exists in common memory. This function also retruns the type handle if the type is defined.

call: istat = CM_FIND_TYPE(name, h_typ)

args: name - in byte array(32) : containing type name, NUL filled.

h_typ - in integer : handle of type

refs: CM_SIMJOIN
SYS$CLREF
SYS$WAITFR
SYS$SETEF

name: CM_FIND_VAR

func: CM_FIND_VAR looks a variable up in common memory, given a name. If the variable does not exist, then it is created. Newly created variables are allocated from the local heap storage implemented as part of the shared common area. Variables are initialized to either binary zero or the value contained in the keyed file.

call: call CM_FIND_VAR(name, iptr, typ)

args: name - in character*32 : name of variable. (Note: passed by DESCR)

iptr - out integer : pointer to variable description in tables.

typ - in integer : pointer to type info arrays.

refs: SYS$CLREF
SYS$SETEF
SYS$WAITFR

name: CM_ISTHERE

func: logical function which returns .true. if the given variable currently exists in common memory. This function also returns the internal pointer to the arrays of variable type information.

call: istat = CM_ISTHERE(name, typ, iptr)

args: name - in byte array(32) : containing variable name, NUL filled.

typ - out integer : pointer to type information arrays. Pointer is stored on the first write sequence of that variable.

iptr - out integer : pointer to variable in internal tables.

refs: none

name: CM_GET_NAMES

func: this routine may be called repeatedly to retrieve the names of all variables currently in common memory. It is the caller's responsibility to maintain a pointer used by this routine between calls. On the first call to search the name table, nptr must be 0. This instructs the routine to start at the top of the name table. The caller should continue calling (without changing nptr) until nptr is again returned to 0.

call: call CM_GET_NAMES(name,typ,ntpr)

args: name - out byte array(32) : variable name, NUL filled.

typ - out integer : pointer to type information arrays.

nptr - inout integer : internal pointer used by CM_GET_NAMES (refer to function).

refs: none

name: CM_GET_TYPES

func: given a handle in the arrays of type information, this routine returns the values for the type's size, class, and subtype.

call: CM_GET_TYPES(thandle, size, class, subtype)

args: thandle - in integer : handle of the type, pointer in type information arrays.

size - out integer : size of variable in bytes.

class - out integer : type class (0 = Basic, 1 = Alias, 2 = Array, 3 = Structure).

subtype - out integer : pointer to more detailed type information in type-info arrays.

refs: none

name: CM_GET_ARRAY_INFO

func: this routine is passed a handle in the arrays of array type information, and returns the number and type of elements in the array.

call: call CM_GET_ARRAY_INFO(arr_ptr, elm_typ, num_elms)

args: arr_ptr - in integer : handle to array type information arrays.

elm_typ - out integer : type of array element.

num_elms - out integer : number of array elements.

refs: none

name: CM_GET_STRUC_INFO

func: this routine is passed a handle in the arrays of structure information, and returns the number of

elements and a pointer to the element list.

call: call CM_GET_STRUC_INFO(str_ptr, elm_list, num_elms)

args: str_ptr - in integer : handle to structure type information arrays.

elm_list - out integer : pointer to list of structure components.

num_elms - out integer : number of structure components.

refs: none

name: CM_GET_COMPONENT_INFO

func: this routine is passed a handle in the arrays of structure component information, and returns the name and type of the component.

call: call CM_GET_COMPONENT_INFO(eptr, ename, etype)

args: eptr - in integer : handle to component information arrays.

ename - out byte array (32) : name of the component, NUL filled.

etype - out integer : handle to the component type.

refs: none

name: CM_SNAP_SHOT

func: this routine makes a snap shot dump of common memory. (Note: file is written on fortran logical unit 3.)

call: call CM_SNAP_SHOT(filnam, fsm_name, fsm_handle, ierr)

args: filnam - in byte array(32) : file name to be used

for snap shot file.

fsm_name - in byte array(32) : FSM name.

fsm_handle - inout integer : FSM handle.

ierr - out integer : error flag which is normally 0.

refs: CM_GET_TIME
CM_READ_REQUEST
CM_GET_NAMES
CM_READ_DONE
SYS$GETTIM

name: CM_ENTER_ALIAS

func: this routine enters an alias for an existing variable. Aliases may be used whenever a variable is read or written.

call: istat = CM_ENTER_ALIAS(alias_name, actual_name)

args: alias_name - in byte array(32) : alias name of variable, NUL padded.

actual_name - in byte array(32) : actual name of variable, NUL padded.

refs: CM_ISTHERE
CM_REMOVE_ALIAS
SYS$CLREF
SYS$WAITFR
SYS$SETEF

name: CM_REMOVE_ALIAS

func: this routine removes an alias from the common memory.

call: call CM_REMOVE_ALIAS(alias_name)

args: alias_name - in byte array(32) : alias name to be removed from common memory tables.

refs: SYS$SETEF
SYS$CLREF
SYS$WAITFR

name: CM_SIMJOIN

func: this procedure performs all initializations necessary for a process to enter into an emulation run. Specifically, it maps the shared common area, or creates and initializes it. In addition, this routine associates cluster event flags, opens all files for data logging, and performs variable initializations. Also, it initiates process-wide statistics gathering, and writes initial logging information to the logging files. Code in this routine uses the following logical names:

AMRF - keyed access file for common memory initialization.

username - the user's login name is used as the logical name for the mapped global section as well as for the cluster name for the event flags.

AMRFSIM.TMP - file name for section file (only used when page file is full).

imagename.LOG - logging file name.

call: call CM_SIMJOIN( )

args: none

refs: CM_GETUSR
CRBUFMAP - macro routine which performs specialized file opening. (Routine is unneeded when emulation is run on VMS versions 3 or greater.)
CM_GETIMAGE
CM_SIMLEAVE
CM_TYPERR
CM_SHRLOCAL$INIT
SYS$WAITFR
SYS$SETEF
SYS$MGBLSC
SYS$CRMPSC

SYS$ASEFC
SYS$TRNLOG
SYS$DCLEXH
LIB$INIT_TIMER

name: CM_SHRLOCAL$INIT

func: this is the data initialization block for the process-local common area of an emulation process.

call: None available

args: None available

refs: None

call: CM_SIMLEAVE

func: this is the exit handler for an emulation process. It is called on image exit by the system exit handler (never by the user). This routine terminates read and write access if they were enabled, writes summary statistics to the logging file, closes the logging and keyed access files.

call: None available

args: status - in integer*4 : exit status passed by system.

refs: CM_READ_DONE
CM_WRITE_DONE
LIB$STAT_TIMER
SYS$CLREF
SYS$SETEF

name: CM_GETUSR

func: this routine retrieves the username from the job process table.

call: call CM_GETUSR(usrnam)

args: usrnam - out character*(*) : username from

the table of processes.

refs: SYS$GETJPI

name: CM_GETIMAGE

func: this routine retrieves the image filename from the job process table.

call: call CM_GETIMAGE(imagename)

args: imagename - out character*(*) : image filename from process table.

refs: SYS$GETJPI

name: CM_GETPRCNAM

func: this routine retrieves the process name from the job process table.

call: call CM_GETPRCNAM(procname)

args: procname - out character*(*) : process name from process table.

refs: SYS$GETJPI

name: CM_TYPERR

func: this routine types out a system error message given the system error code.

call: call CM_TYPERR(istat)

args: istat - in integer : system error message code.

refs: SYS$GETMSG

Documentation from file : SHAREOUT.PRX

name: SHAREOUT.PRX

func: SHAREOUT.PRX implements delayed writing to common memory. Calls to this module are issued by both the PARSER (using subroutine CM_STORE_OUTPUT) and the BUILDER (using subroutine CM_DUMP_OUTPUT). Since the emulation allows multiple state machines per process, individual machines could not write their outputs to common memory at the end of the table. This would have resulted in the next state machine in the same process seeing inconsistent variables in common memory, i.e., not all variables written on the same tick. To circumvent this problem, PARSER generates calls to CM_STORE_OUTPUT. The CM_STORE_OUTPUT routine constructs a copy of the written variable in dynamic (Praxis ALLOCATE) memory. All variables are written to this list from all state machines within a given VMS process. After all state machines within a process have been invoked, the main module, constructed by BUILDER, searches for all variables which have a zero (0) write delay. These variables are written to common memory (using CM_WRITE_VARIABLE) and their dynamic memory FREE'd. Any variable with a non-zero write delay has the delay decremented by one tick, and remains on the list.

Subroutines in SHAREOUT.PRX

name: CM_STORE_OUTPUT

func: this routine stores common memory write requests in dynamic memory. Variables are stored with their handle, delay, and value on a dynamic, in_memory list.

call: call CM_STORE_OUTPUT(vhandle, variable, vsiz, delay)

args: vhandle - in integer : handle of the variable being stored. Handle is returned by the routine CM_OPEN_VARIABLE in SHRMEM.FOR.

variable - in general : value of variable.

vsiz - in cardinal : size of variable in bits.

delay - in integer : the delay, in ticks, before writing value to common memory.

refs: none

name: CM_DUMP_OUTPUTS

func: this procedure scans the dynamic list constructed by CM_STORE_OUTPUT and writes all variables whose time has come into common memory. It also removes these variables from the list and releases their storage. Any variable whose delay count has not been reached is left on the list and the delay is decremented.

call: CM_DUMP_OUTPUTS( )

args: none

refs: CM_WRITE_REQUEST
CM_WRITE_VARIABLE
CM_WRITE_DONE

name: DISPLAY

func: DISPLAY is the main observation tool of the emulation. DISPLAY runs as an independent process which has access to common memory. The user may interact to display variables, change the speed of the emulation, single step the emulation, and dump snap shot files. DISPLAY will run on any terminal supported by the DEC VMS run time library of scope commands. The user should start all other emulation processes (if they are being run as subprocesses) prior to running DISPLAY. When DISPLAY is activated, it clears the screen and waits for the user to type any character, except "E" and "D" which are reserved for special functions, to begin the display process. The user has the following single character commands available to him.

E - Enter new variable(s) for display
D - Delete variable(s) from the display
M - scroll display area up or down a given number of lines
up-arrow - scroll display area up
down-arrow - scroll display area down
right-arrow - shift display area right
left-arrow - shift display area left
C - Change the speed of the emulation
S - Single step the emulation
G - Go, resume continuous operation (used after S)
Q - Quit or exit DISPLAY program
L - Log a snap shot dump of common memory
H - Help (which essentially contains this file)

A brief description of the commands are:

D - Enter variable name(s) to be added to the display. This variable name may contain wildcard (*) characters to select whole classes of variables. Common memory is searched for all variables meeting this specification and the indicated variables are added to the bottom of the display.

NOTE: The search is performed at the time the the "enter variable" command is issued and not on every subsequent time tick; hence, variables which subsequently appear in common memory which might meet a previous specification will not be displayed!

D - Delete variables from current display; wildcards are accepted.

M - Move the display window, and thus, the display of variables up or down a designated number of lines. It accepts any non-zero integer, but will not move the variable display pass the uppermost or lowermost line in the variable list. Scrolling of variables can also be accomplished using the up or down arrows, which will scroll the display one line in the respective direction. To move the display right or left, the right and left arrow may be

used.

C - Change the speed of emulation. The command accepts a single number specifying the ratio of wall clock time to emulated time, i.e., entering a 10 causes the emulation to run at 1/10th real time. Entering 0 disables timing of the emulation and allows the emulation to run at its maximum rate.

S - Single steps the emulation. One common memory read/write cycle is executed for each time the 'S' key is pressed. Between each single step, the emulation is STOPPED. The user may examine or delete variables while in single step mode. Continuous operation may be resumed at speed last specified (if none specified, emulation runs at maximum speed) by the 'G' (Go) command.

G - The 'Go' command resumes continuous operation after single stepping through the emulation.

L - Log a snap shot of the values in common memory at the time the 'L' key is pressed. The command requests an output file name for the memory dump. The emulation is temporarily halted while the current contents of common memory are written to a logging file.

H or ? - Prints a listing of available commands.

Q - Quit cause the display program to exit. It is somwhat safer to exit the display program with this command than to arbitrarily use ^Y. While every attempt has been made to protect common memory from dead locks, one can never be quite sure.

refs:

LIB$PUT_SCREEN
LIB$ERASE_LINE
LIB$ERASE_PAGE
LIB$GET_EF
LIB$SET_SCROLL
LIB$SET_CURSOR
LIB$UP_SCROLL
LIB$DOWN_SCROLL
CM_GET_TIME

```
CM_READ_REQUEST
CM_READ_VARIABLE
CM_READ_DONE
CM_WRITE_REQUEST
CM_WRITE_DONE
CM_ISTHERE
CM_GET_NAMES
CM_GET_TYPES
CM_GET_ARRAY_INFO
CM_GET_STRUC_INFO
CM_GET_COMPONENT_INFO
CM_SNAP_SHOT
SYS$TIMR
SYS$WAITFR
ISBYTE  (in library, BP_LIBRARY)
GETCH                  "
PUTCH                  "
CVNTIM                 "
```

name: SIMLIST.PRX

func: SIMLIST uses the logging file produced whenever an emulation is run. First, all user-defined types are listed, then, each variable is listed whenever their values change during the emulation. SIMLIST uses Fortran routines to open and close the log file, and to read a record from the log file into a character buffer record.

Format: SIMLIST is defined as the foreign command

```
$ SIMLIST :== $HCSE_LIBRARY:SIMLIST
```

Command syntax:

```
$ SIMLIST file
or $ SIMLIST file.log
```

Output goes to logical unit SIM_OUTPUT if the logical name exists and can be opened. Otherwise, output goes to SYS$OUTPUT.

call: main module; run as a VMS foreign command, i.e, $ SIM file.log.

refs: READREC.FOR (found in CM_LIBRARY)

name: SUMMARY.PRX

func: SUMMARY reads a log file, lists all the statistics and summarizes the variable transitions observed. It tabulates transition information for variables including the variable's last value, the number of transitions for all types, the minimum and maximum values for numeric types, and the amount of time spent at each value for string types.

Format: SUMMARY is defined as the foreign command

```
$ SUMMARY :== $HCSE_LIBRARY:SUMMARY
```

Command syntax:

```
    $ SUMMARY file
or $ SUMMARY file.log
```

Output goes to logical unit SUM_OUTPUT if the logical name exists and can be opened. Otherwise, it goes to SYS$OUTPUT.

call: main module; run as a VMS foreign command, i.e., $ SUMMARY simrun.log.

refs: READREC.FOR (found in CM_LIBRARY)

FEDERAL INFORMATION PROCESSING STANDARD SOFTWARE SUMMARY

01. Summary date: Yr. 85 | Mo. | Day
04. Software date: Yr. 85 | Mo. 11 | Day 5

02. Summary prepared by (Name and Phone)
Cita Furlani, 921-2461 area code (301)

05. Software title
Hierarchical Control System Emulation Programmer's Manual

06. Short title

03. Summary action
New [X] Replacement [ ] Deletion [ ]
Previous Internal Software ID

07. Internal Software ID

08. Software type
- [X] Automated Data System
- [ ] Computer Program
- [ ] Subroutine/Module

09. Processing mode
- [X] Interactive
- [ ] Batch
- [ ] Combination

10. Application area

General
- [ ] Computer Systems Support/Utility
- [X] Scientific/Engineering
- [ ] Bibliographic/Textual
- [ ] Management/Business
- [ ] Process Control
- [ ] Other

Specific
Industrial Control Design

11. Submitting organization and address
U. S. Department of Commerce
National Bureau of Standards
Bldg. 220 - Room A-127
Gaithersburg, MD 20899

12. Technical contact(s) and phone
Cita Furlani 921-2461 area code (301)

13. Narrative
The Hierarchical Control System Emulation is a collection of computer programs written in the high-level Praxis language for use on a Digital Equipment Company VAX 11/780 TM processor under the VMS TM operating system. These programs allow the user to write, debug, and concurrently emulate modules of a hierarchical control system and to simulate the physical plant which is controlled. The emulation executes in real time and interactive display and data logging capabilities are included. The emulation is intended as a computer-aided control system design tool for the NBS Automated Manufacturing Research Facility. The Programmer's Manual provides documentation of the design of the emulation code and the emulation programs themselves; it is intended for the system programmer rather than the user.

14. Keywords
Automated manufacturing; automatic control; hierarchical control system; computer-aided design; computer-aided manufacturing simulation.

15. Computer manuf'r and model	16. Computer operating system	17. Programing language(s)	18. Number of source program statements
DEC VAX 11-780	VMS, Vers. 2.7	Praxis, Fortran	
19. Computer memory requirements	**20. Tape drives**	**21. Disk/Drum units**	**22. Terminals**
1 Mbyte	None	System disk required	VT52, VT100 or equivalent

23. Other operational requirements
None

24. Software availability
Available [ ] Limited [X] In-house only [ ]
For government use only.

25. Documentation availability
Available [X] Inadequate [ ] In-house only [ ]
NTIS

26. FOR SUBMITTING ORGANIZATION USE

Standard Form 185
1974 July
U.S. Dept. of Commerce—NBS
(FIPS Pub 30)

NBS-114A REV. 2-8C

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NBSIR 85-3156	2. Performing Organ. Report No.	3. Publication Date May 1985

4. TITLE AND SUBTITLE

Hierarchical Control System Emulation
Programmer's Manual

5. AUTHOR(S)

Cita Furlani (Editor)

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

10. SUPPLEMENTARY NOTES

Previously published as NBS-GCR-82-414 - NTIS # PB83-137059.

[X] Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

The Hierarchical Control System Emulation is a collection of computer programs written in the high-level Praxis language for use on a Digital Equipment Company VAX 11/780™ processor under the VMS™ operating system. These programs allow the user to write, debug, and concurrently emulate modules of a hierarchical control system and to simulate the physical plant which is controlled. The emulation executes in real time and interactive display and data logging capabilities are included. The emulation is intended as a computer-aided control system design tool for the NBS Automated Manufacturing Research Facility. The Programmer's Manual provides documentation of the design of the emulation code and the emulation programs themselves; it is intended for the system programmer rather than the user.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

Automated manufacturing; automatic control; hierarchical control systems; computer-aided design; computer-aided manufacturing simulation.

13. AVAILABILITY

[X] Unlimited
[ ] For Official Distribution. Do Not Release to NTIS
[ ] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
[X] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES
~7

15. Price
$8.50