NASA Technical Memorandum 89015

FTMP Programmer's Manual

Frank E. Feather

Carlos A. Liceaga

Peter A. Padilla

September 1986

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Table of Contents

List of Figures

FTMP Programmer's Manual

The Fault Tolerant Multiprocessor (FTMP) computer system was constructed using the Rockwell/Collins CAPS-6 processor. It is installed in the Avionics Integration Research Laboratory (AIRLAB) of NASA Langley Research Center. It is hosted by AIRLAB's System 10, a VAX 11/750, for the loading of programs and experimentation. The FTMP support software resides on the IBM 4381 computer of the Business Data Systems Division of Langley Research Center. This support software includes a cross compiler for a high level language called Algol Extended for Design (AED), an assembler for the CAPS-6 processor assembly language, and a linker. Access to this support software is through an automated remote access facility on the VAX which saves the user of the burden of learning how to use the IBM 4381.

This manual is a compilation of information about the FTMP support environment. It explains the FTMP software and support environment along with many of the finer points of running programs on FTMP. This will be helpful to the researcher trying to run an experiment on FTMP and even to the person probing FTMP with fault injections. Much of the information in this manual can be found in other sources; we are only attempting to bring together the basic points in a single source. If the reader need any points clarified, there is a list of support documentation in the back of this manual.

This the third edition of this manual. The first edition was written by Frank Feather in the summer of 1984 and consisted of a shell of sections on how to use the facilities at AIRLAB and clues on running FTMP programs. The first edition was really a draft of the second edition but was getting extensive use before the full manual was released. This first manual was used around the FTMP station at AIRLAB and had nearly as many penciled in comments as typed words. A user's guide to FTMP, written by Carlos Liceaga, also existed at AIRLAB. That user's guide was merged with the first edition to form the second edition of FTMP Programmer's Manual.

In spring 1985 the programming process was greatly simplified by the installation of a program by Peter Padilla for doing remote compilation and linking from the VAX. This rendered one whole chapter of the second edition obsolete, since the FTMP user no longer had to deal with the IBM. The major changes in this edition of the manual is the description of the new programming environment and the deletion of obsolete sections.

There are several people who contributed to this manual. First, the AIRLAB staff, who helped with any problems we had with the VAX or FTMP. Two of the initial researchers of FTMP at AIRLAB are Matt Reilly, who wrote the a program called "CONNECT" to connect the VAX to the IBM, and Ed Clune, who initiated the authors into the FTMP work environment and wrote some of the first experiments for

FTMP. Through Ed's help and observations we began much of the initial work on the manual. Also, the very first guinea pigs to the manual, Ann Marie Grizzaffi and Edward Czeck, both of whom were able to get their first program running much more quickly (1 day) than the first programs that Frank Feather, Ed Clune and Matt Reilly wrote (literally months!). Finally, we wish to thank all of those user's of FTMP, Mike Woodbury and researchers at University of Michigan.

1. Programming FTMP

The FTMP software is written either in a high-level language called AED or in the CAPS-6 assembly-level language. There are no facilities on the FTMP itself to compile or assemble programs. Instead, source programs must be converted into the CAPS machine language on a mainframe computer that supports AED. At AIRLAB, that mainframe is an IBM 4381. To create tasks for the FTMP the programmer must create an AED program and a linker command file on a VAX11/750 (System 10 at AIRLAB) and use the remote compile and link facility (Section 1.2.1) to cross-compile, assemble and link the program to produce a load file. The load file is automaticly downloaded from the IBM 4381 to the VAX by the remote access facilities and must then be downloaded to the FTMP. The support computers for FTMP and their interconnection are illustrated in Figure 1-1

The rest of this chapter mainly discusses the process of creating a program for FTMP using the cross-compiler, assembler, and linker. It will also review some of FTMP's operating system structures and what these structures mean to the programmer trying to add tasks. Finally, we shall discuss the downloading process so the program task can be executed on FTMP.

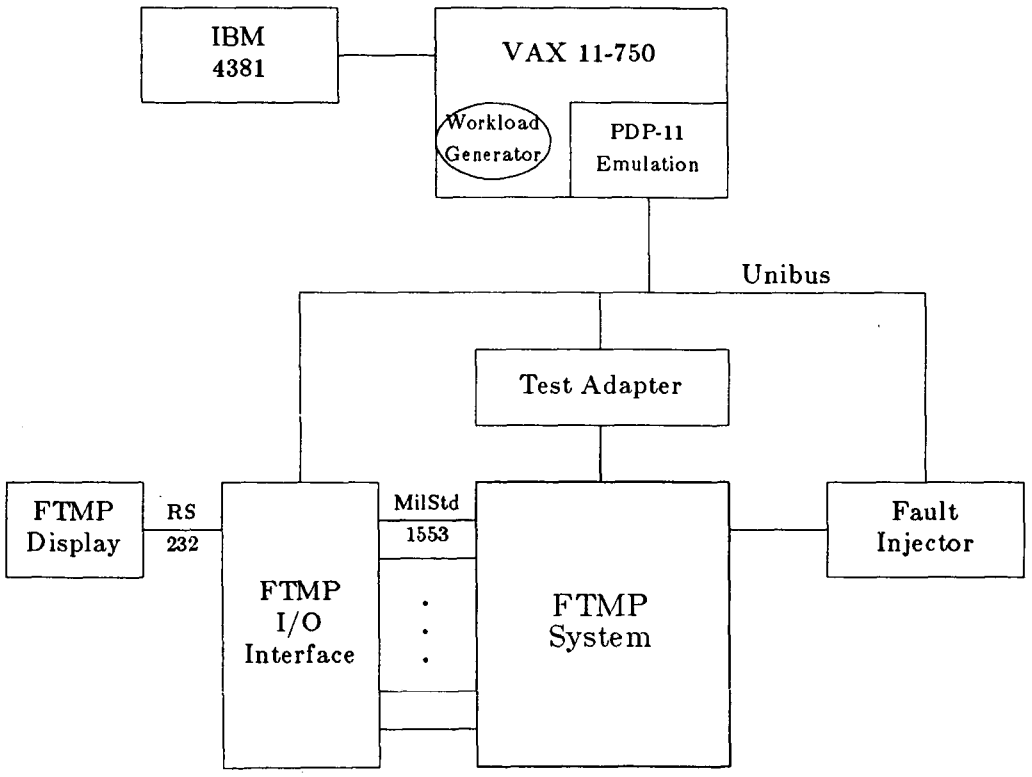


Figure 1-1: FTMP Support Environment

1.1. Overview

The reader is undoubtedly familiar with the compile/assemble/link process of creating a program (Fig. 1-2). In review, the programmer creates a task or modifies an existing task written in a high-level language (AED) for execution on the target computer (FTMP). This code is compiled to produce a relocatable object code module. If any assembler files are created or modified, they too must be assembled down to relocatable object code. Finally, the new object modules must be linked in with existing object modules to produce a load module for downloading.

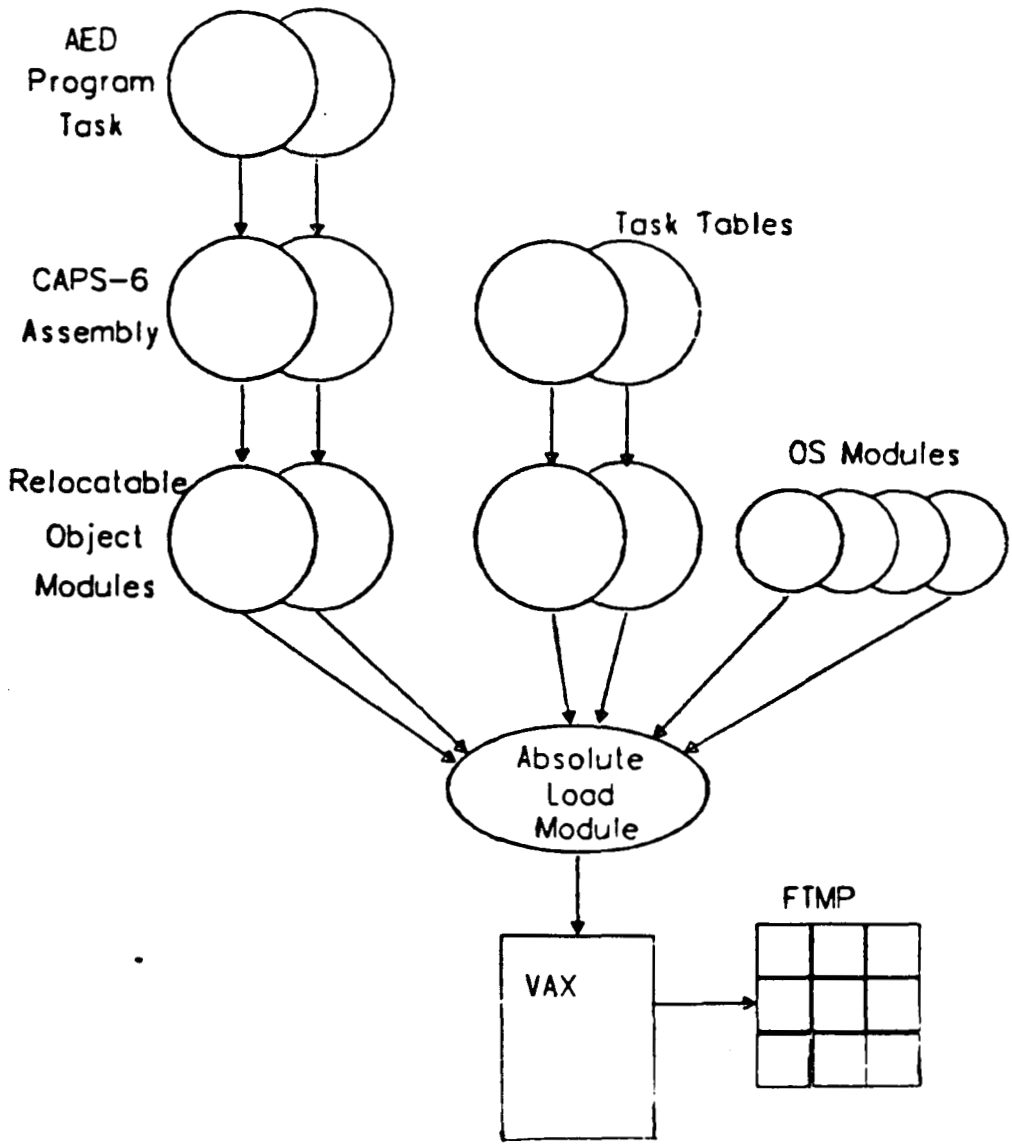


Figure 1-2: Compile/Assemble/Link process

On FTMP, the full process of creating or modifying a program task can be quite complicated. Besides compiling an AED program, the user has to map out where variables and code go (in local cache memory or shared system memory), and must modify system tables to include the task in the system task

structure. After this, the user must re-assemble operating system tables and link everything together. We will be referring to Figure 1-2 throughout the manual to remind the reader of the steps to preparing a program.

1.2. Accessing the IBM

The previous edition of the manual explained the process of editing, compiling and linking files on the IBM 4381 shown in Figure 1-1. FTMP users were required to learn this system in addition to the VAX and FTMP. Much of that process has now been automated so the user no longer needs to log onto the IBM. Instead, the FTMP software now resides on the VAX. Through VAX command files, FTMP programs are uploaded to the IBM, processed (compiled, assembled or linked) and the results are downloaded back to the VAX. The next two sections discuss the file structure of the FTMP software and the commands for compiling, assembling and linking FTMP files.

1.2.1. File Structure

The FTMP software now resides in a VAX-11/750 in several VMS directories on the DISK$MO cluster disk in AIRLAB. The directories are organized similar to the way they where on the IBM system. There are three main directories: FTMP, CMU, and MUA. Each directory has the following subdirectories:

1) ASM.DIR
2) AED.DIR
3) LINKLIST.DIR
4) LOAD.DIR
5) LINK.DIR
6) LIST.DIR

The subdirectories contain the source code, both CAPS assembly (ASM.DIR) and AED (AED.DIR), the FTMP linker command files (LINK.DIR), the load modules (LOAD.DIR), the linker listings (LINKLIST.DIR) and the compiler and assembler-generated listings (LIST.DIR). IBM Job Control Language (JCL) files are stored in a different directory and users are not allowed access to it. The directory contains the JCL files sent by the VAX system to the IBM to execute according to commands issued by the users on the VAX.

All files are standard VMS files. The type of a file reflects its content: AED source files are named *filename*.AED;xxx (xxx = three digit version number), CAPS assembly files are named *filename*.ASM;xxx, link command files are *filename*.LIN;xxx, and load modules are *filename*.LOA;xxx.

Access to the FTMP, CMU, and MUA directories is limited to users with the 175 group UIC. The files can be edited with any VMS editor and sent to the IBM for processing. Access to the FTMP directory is limited to read only. If you need a file from this directory you should copy it to the corresponding CMU

or MUA directory and edit it there.

1.3. VAX Commands to Compile, Assemble, and Link

Several commands have been implemented on the VAX-11/750 system 10 in AIRLAB to support the processing of the FTMP software. Once you have edited a file and it is ready to be compiled, you need only type one command and the file name. The system will perform all the file handling and communications with the IBM for the user. The commands are:

a) to compile *filename*.AED file:
   1) CMUAED *filename* (to access CMU directory)
   2) MUAAED *filename* (to access MUA directory)

b) to assemble *filename*.ASM file:
   1) CMUASM *filename* (to access CMU directory)
   2) MUAASM *filename* (to access MUA directory)

c) to link using *filename*.LIN linker command file:
   1) CMULINK *filename* (same as above)
   2) MUALINK *filename* (same as above).

By issuing these commands, you tell the system to access the source directory, read the specified file (the file name should be of the proper type, i.e. AED, ASM, LIN and/or LOA), and send that file with the proper JCL file to the IBM for execution. You do not need to edit JCL files, the system takes care of this. Note that the object files produced from a compilation or assembly are stored in the IBM system; therefore it is not necessary to send them every time a link is performed. The listing generated by the compiler and/or assembler is sent back to the VAX and is stored in either [CMU.LIST] or [MUA.LIST] directory. The load modules produced from a link are sent back to the VAX and are stored in either [CMU.LOAD] or [MUA.LOAD].

When issuing a command, if the source file is not the highest version, you have to specify the complete file specification except for the directory (i.e. *filename*.type;version). The system looks for the specified file in the default directory (e.g. [CMU.AED] or [MUA.AED]), and if the file is not there or if the *filename* is not of the proper type an error message will be displayed on the terminal.

After the command had been issued the user does not need to wait for the compilation, assembly or link to finish. A mail message will be sent to that user notifying him that the processing on the IBM had been completed successfully. A fatal error message in the mail means that an error occurred in the communications with the IBM. In this case you should notify AIRLAB personnel before trying again.

The communication system used to send jobs to the IBM can handle only one job at a time so several users should not try to compile/assemble or link simultaneously.

1.4. AED

The purpose of this manual is not to teach AED; the FTMP user familiar with other block structured languages like PASCAL or ALGOL, can pick up AED quickly by reading the AED manual [13] and looking at AED code. There are, however, a few things to remember about using AED for creating tasks on FTMP.

1. End each task with a HALT statement (or RESUME if you have SYNONYMS HALT=RESUME at the beginning of the program). If you forget this statement your task will run once and never return control to the task scheduler, thus stalling the FTMP computer.

2. In the FTMP AED compiler, parameters are either pass by reference or pass by value (differing from standard AED compilers in which all parameters are passed by reference). Recall that with pass by reference the **address** of the variable is passed as a procedure argument rather than its **value**, as happens with pass by value parameters. In CAPS software, pass by value arguments are more efficient than reference arguments. Much of the AED documentation is a little sketchy about how to define parameters for your procedures, and how to declare external procedures and the type of parameters they use (i.e. pass by reference or by value). The following summary should clarify things.

   - When defining procedure parameters, prefix the variable type statement with "INPUT." to make it pass by value; otherwise it is pass by reference.

   - You must state in the declaration of an external procedure the type of parameters it takes. These parameters are declared just like when defining a procedure. If you do not state the parameter types for an external procedure, parameters are assumed to be pass by reference of ANY data type.

   You can also abbreviate parameter declarations (i.e. A instead of ANY). Appendix II contains a list of parameter declaration argument types and their abbreviations. The following program segment illustrates the above points:

```
      .
      .
      .
EXTERNAL OTHER;          ... Defined in another module //
INTEGER  OTHER;          ... but used in this module.  //

PROCEDURE HREAD;               ... External procedures //
PROCEDURE RD(A,A,II);

DEFINE PROCEDURE EXAMPLE (FIRST, SECOND, THIRD, FLAG)
   WHERE INTEGER FIRST;                  ... Reference //
         INPUT.INTEGER SECOND, THIRD;    ... Value     //
         BOOLEAN FLAG      TOBE          ... Reference //
   BEGIN
   OWN INTEGER I;         ... Stored on stack (dynamic) //
       INTEGER J;         ... Static local Variable     //
   EXTERNAL R4.CACHE;     ... Task Data Block           //
   INTEGER ARRAY R4.CACHE;

   EXTERNAL MINE;           ... Defined and allocated  //
   VOLATILE.INTEGER MINE;   ... in this module but     //
   PRESET MINE=0;           ... available externally.  //
      .
      .
      .
```

In this example, HREAD and RD are external procedures. Procedure HREAD has reference parameters (default since no parameter declaration), and RD has two ANY reference parameters and one integer value parameter. Note that for HREAD we don't have to state the number of parameters it uses if we choose not to. Procedure EXAMPLE is defined to take four arguments; the first one is an integer reference, the next two are value parameters, and FLAG is a boolean reference parameter.

3. **Declaring Variables**. Variables are stored differently depending on how they are declared. The following is a summary of variable storage allocation:

   - **Dynamic Local Variables.** Local OWN variables are declared inside a procedure body. Room for such variables is dynamically allocated on the stack when the procedure is called and subsequently deallocated when the procedure is exited. The value of a local OWN variable is undefined with each call of the procedure. In the above example, I is such a variable. Stack variables are the most efficient in terms of access time since CAPS-6 is a stack machine.

   - **Static Local Variables.** Any variable declare within a procedure body, not declared OWN or EXTERNAL will be constructed as a **static** local. These variables can only be used by the declaring procedure and nested subprocedures and are allocated in cache. Actual location of these variables depends on the linker, although presently linker commands put them at the bottom of the rate group's stack page. Although these variables are undefined when you first enter the task or procedure, recursive calls will retain these variables' values since they are statically allocated in cache (unlike OWN variables where each recursive call allocates a new set of dynamic local variables). In the above example, J is a static local variable.

   - There is a type of local variable that retains its value between task iterations called a **task data block** variable. Data block variables are loaded into cache from system memory by the dispatcher before the task starts, and moved back to system memory when the task finishes. By this scheme, task data block variables retain their value no matter which processor a task is run on. The data block variables are stored in an array called Rx.CACHE, where x is the task rate group (1,3,4). This array is located at the bottom of this rate's stack page (the stack grows from the top of the page down), and must be declared EXTERNAL to access. The size of this array depends on how you set your stack limit in the linker command file and on the Data Control Block for this task in the system tables (see section 1.5.3). Pages 76-78 of FTMP manual, volume II [11], describe the task data block.

     When using **data block** variables and **static local** variables together, the user should make sure that cache storage for these two do not overlap since both are usually at the bottom of the appropriate rate group's stack page. Also, be sure to adjust the stack limit as you add more storage for either of these variable types. All of these adjustments are controlled by linker commands and modification of task tables (see section 1.5.1, Linking Things Together).

   - **Volatile Variables**. VOLATILE variables are much like static local since they are defined and statically allocated with their declaring module. The major differences are that a Volatile variable *must* be PRESET and can be declared EXTERNAL if the user wishes to make it available to other procedures.

   - **External Variables**. With an EXTERNAL statement, the module can declare for use

a variable which is statically allocated in another module, or can make a VOLATILE variable available to other procedures. EXTERNAL variables are essentially system global and available to all procedures. THERE IS NO WAY TO TELL FROM ITS DECLARATION IF AN EXTERNAL VARIABLE IS IN LOCAL (CACHE) MEMORY OR IN SYSTEM MEMORY. This constraint depends on how you set up your linker commands and how you access the variable (i.e. with system bus service routines or regular assignment). As a rule, all system memory variables are defined in the tables file ([ASM]TABLE.ASM) -- all others like VOLATILE should be defined in cache. Finally, only a process running in PRIVILEGED can use the bus service routines to access system variables.

- **Constants.** Any variable that is PRESET and not declared VOLATILE will be constructed as a constant.

As noted above, all system memory variables are defined in the tables file (see section 1.5.3), which is a CAPS assembly file. Therefore, a brief explanation of assembly level declarations is warranted. Fortunately, defining variables at the CAPS assembler level is a little easier than in AED. There are two basic rules for declaring variables:

a. If the user wants to make a local variable available to other modules, he simply puts the statement

ENTRY *variable*

at the top of the file. Of course, *variable* must be used as a label somewhere in the CAPS file.

b. The statement

EXTERNAL *variable*

declares that *variable* is defined in another file but is used in this file.

ENTRY and EXTERNAL variables will be in the link map (LFMTN.CMU.LINKLIST). Any variable in a CAPS assembly file not declared with an ENTRY or EXTERNAL statement are local and undefined outside their CAPS file.

1.5. Linking Things Together

Once your AED program task compiles correctly it must be incorporated into FTMP's task structure and linked with existing relocatable object modules to produce an absolute load module. This section explains such a process.

1.5.1. Link Files

Linker command files are located in the subdirectory LINK.DIR. Through linker commands the user specifies where, and in which of FTMP's memories (local cache or system) to put object code, where variables reside, and where to put the runtime stacks and Operating System structures. Any time the user creates or modifies a program task, that task must be linked in with all of the Operating System modules. The link file also specifies Operating System modules. The next few paragraphs outline the linker commands.

SEG: A link file can put code in either local PROM or Global memory with the following statements:

```
SEG PROM
SEG Mxxx
```

Relocatable object modules following the first statement are directed into the FTMP PROM memory. PROM code records are prefaced with the letter **P** in the absolute load module. The PROM address space is 0 to 1FFF. and contains frequently used routines like dispatchers and the O.S. kernel. If you do change PROM, all LRU's (Line Replaceable Units) you are going to use must be reprogrammed. We do not recommend changing PROM unless you are sure that's what you want.

The second statement

```
SEG Mxxx
```

puts the code for object modules following it into FTMP system memory offset by location xxx (usually xxx=0000). Code in main memory is automatically paged into local cache memory as needed. These records are prefaced with the letter S in the load module.

Finally, if you want to replace the PROM board in a LRU with a RAM board, you can either use the statement

```
SEG RAM
```

in the link file or can change all of the P's prefixing PROM code in the load file to R's.

ORG: The "ORG" linker command specifies where read/write data and stacks, code, and read only data go. The format of the ORG command is

```
ORG "xxxx",<y>
```

where xxxx is the hex offset and <y> is defined as follows:

```
0     Data and stacks
1     Code
2     Read only data
```

Thus the statement

```
ORG "2800",1 ;
```

starts putting code of relocatable modules following this statment into memory location hex(2800) of the absolute load module.

IN: The IN command tells the linker to include the given relocatable object module in the link. Thus,

```
IN LIB(SCC)
```

will link in the object module that contains the system configuration control (SCC) routines. You will need an IN statement to include the module that has your program task.

EQU: The EQU command binds a previously undefined variable to the value or variable name following the EQU. Thus the statement

R1.TASK1 EQU SCC ;

sets R1.TASK1 to be procedure SCC. There is an **assembler** pseudo-op EQU which is equivalent in meaning and format to the **linker** EQU command.

Generally, you should not have to change or add SEG or ORG commands when building a link file -- the explanation of these commands is only included so that the user understands the linklist output produced by the linker.

For those of you thoroughly confused by the above discussion, an example is in order.

```
SEG     PROM            ;
                        ;
ORG     "0000",2        ; Read only data area
ORG     "2000",0        ; Read/write data area (2000 to 24FF)
ORG     "0100",1        ; Code area -- loc hex(100) on
                        ;
IN      LIB(REGS)       ;  Register definitions
IN      LIB(KERNEL)     ;  O.S. Kernel
       ...
                        ;
SEG     M000            ;
                        ;
ORG     "2800",1        ; Code area in Main memory -- hex(2800) on
ORG     "2500",0        ; PSD's and STACKS (2500 to 27FF)
IN      LIB(SCC)        ; System configuration control
IN      LIB(SELFTEST)   ; Master Self test
      ...
                        ;
TASK.R11 EQU SCC        ; "TASK.R11", declared EXTERNAL in another
                        ;  is now defined to be the procedure "SCC".
R11.STKLM EQU *+6,0     ;  Stack limit is "2500"+6
                        ;  Top Of Stack is "25FF" (defined in TABLES)
                        ;  (TOS and STKLM are the cache locations)
    ...
                        ;
  ORG   "2700",0        ;  Put R4 task variables at location "2700"
  IN    LIB(EXAMPLE)    ;

TASK.R41  EQU TEST      ;
R41.STKLM EQU *+6,0     ;  R41 stack limit is "2700"+6
                        ;
   ...
  ORG   0,0             ;  SYSTEM MEMORY IMAGE
  IN    LIB(TABLES)     ;  "TABLES" CONTAINS VARIABLE DEFINITIONS ONLY
  END                   ;
```

The AED linker creates an output file which lists link errors (undefined symbols, multiply defined symbols, etc.) plus contains a load **module memory map** and a **cross reference listing**. The list file is in the subdirectory LINKLIST.DIR.

For those requiring further reference, the linker is discussed briefly in the FTMP Software Manual [11], section 6.3, and discussed more extensively in the CAPS Link Editor User's Guide [7].

1.5.2. Prototype Link File

Understanding the meaning of linker commands is fine, but simple instructions on how to modify an existing link file to create an absolute load module with your task in it would be even better. Fortunately, there is a prototype link file named EXEC.LIN that can be used with few modifications. Make a copy of this file for modification. There are four major things to check and modify if necessary:

1. Decide which System Configuration Control task to use -- SCC or FSCC. SCC is the normal one and FSCC is a special task used when doing fault injection experiments. The "IN LIB(SCC)" statement is at about line 46 of the EXEC link file.

2. Assign your program task a rate group and task number. Look for a statement
   TASK.Rxy EQU <procedure name>
   where x is the rate group (1,3 or 4) and y is the task number. Change <procedure name> to the name of your program task. Usually, your program task will be TASK.R41, .R42, or .R43. If there is not a label for the rate group and task number that you want, you will have to create a *task control block*, a *dummy PSD block*, and (optionally) a *data control block* for your task (see section 1.5.3).

3. Include your program module using an "IN LIB(<module name>)" statement. A good place for this statement is immediately before the statement defining your program's rate group and task number.

4. Check that you include the correct "tables" file that has your system variables and set up task tables. The statement for including the tables is the second to the last statement in the link file.

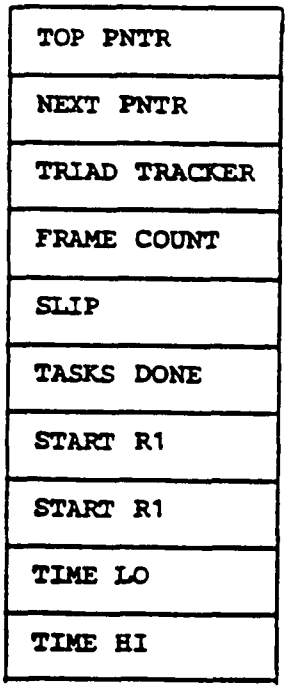
Also, if you are using a lot of local and a data block variables, adjust the stack limit appropriately in the link file. Local and data block variables are at the bottom of the stack page while the stack grows from the top of the stack page down.

Once you have created a link file for your task you should never have to modify the link file again, even if you recompile you task, unless you add more tasks.

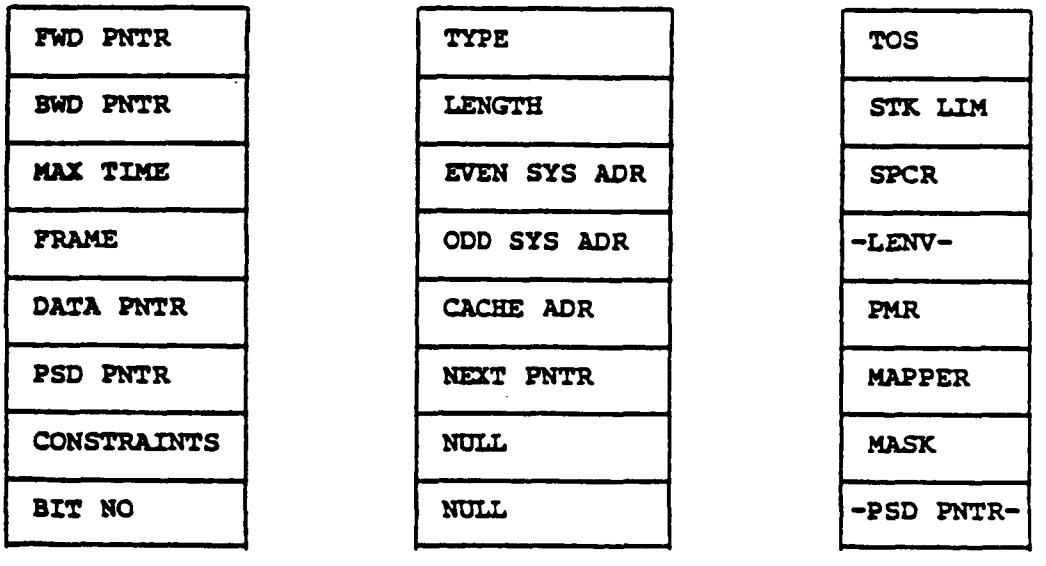
1.5.3. Setting up Task Tables

Through linker commands you specify task names plus the location of code and variables. However, through the task tables you actually link tasks into the Operating System structure. Also, if the user creates any global variables they must be defined in the system tables.

The file TABLES.ASM defines the *system memory* image. Every system variable that the user or operating system uses must be defined in this file since this file essentially maps where everything is



R4 control block.



(a) Task control block (b) Data control block (c) PSD

Figure 1-3: Operating System Task Structures

located in memory. There are two reasons that the user would be interested in the TABLES file:

1. The user's tasks use global variables in system memory (i.e. variables read in with "RD"). TABLES is the system memory image. In this case the user will define his variables at the end of one of the pages, or (more likely) somewhere on pages (hex) F through 28 -- the free system memory pages. Such variables must be declared with an ENTRY statement (usually at the top of the file), and its actual location is marked by the symbol name starting in the first column, followed by a RES or VALUE statement. The link map, located in the subdirectory LINKLIST.DIR, will give the location of the variables.

2. The user has added a task and needs to create a Task Control Block, dummy PSD, and Data control block for the task. In addition, pointers need to be set to link these new task structures in with the other Operating System structures. Figure 1-3 are the operating system task structures. These structures are explained in the FTMP Software manual [11], pages 74-79. It is strongly recommended that the user read these pages of the Software manual before proceeding. Following is a review of the structures. We will particularly elaborate on fields whose implementation differs from the FTMP software manual's explanation.

   R4 Control Block: Implementation is as explained in the FTMP software manual. Field one of the R1 control block, "TOP PNTR", may be changed by the dispatcher while FTMP is running because of task re-ordering due to constraint bits (see explanation of constraint field in task control block below). The names of the blocks in TABLES are R4, R3, and R1.CONTROL.

   Task Control Block:

   Every task on FTMP has a Task Control Block assigned to it. Actual implementation of this block differs significantly from what is described in the FTMP software manual.

   - **FWD PNTR** -- Pointer to next Task Control Block in the task list or NULL if end of list. For R1 tasks, this pointer may be changed by the R1 dispatcher when it reorders the tasks. Also, you can use CTA to change this field while FTMP is running to link new tasks in dynamically (very useful while experimenting and debugging).

   - **BWD PNTR** -- Unused. There is no backward pointer.

   - **MAX TIME** -- Max time, in clock ticks, that the task is allowed to run before it is aborted. One clock tick is 0.25 milli-seconds.

   - FRAME -- Same as in the FTMP Software manual.

   - DATA PNTR -- Pointer to this task's Data Control Block, or NULL if no such block.

   - PSD PNTR -- Pointer to this task's PSD block.

   - **CONSTRAINTS** -- The description in the FTMP software manual has nothing to do with actual implementation. For R4 and R3 tasks, this constraint field is unused. For R1 tasks, this field is 0 or processor triad id number (1, 2, or 3) and specifies the processor triad that the R1 dispatcher should try to run this task on. A zero field (0) indicates no processor triad constraint. If the processor triad is not

up, then the task can run on any processor. The dispatcher does not guarantee running a task on its requested processor triad if other R1 tasks request the same triad. Also, the dispatcher will reorder tasks in system tables if a constrained task has to wait for a processor to become available (the constrained task is slipped down the list if its requested processor is unavailable). Presently, the only task that uses the constraint field is "SCC" (or "FSCC" if using fault injection configuration). "SCC" systematically changes the constraint field so it can run a full round of self-tests on each processor.

- BIT NO. -- Has no meaning since constraint bits are not implemented as specified in the manual.

Data Control Block:

Actual implementation is as described in the FTMP software manual, except that NEXT PNTR is unused. The meaning of the fields are:

TYPE: 0--RD/WRT, 1--RD, 2--WRT.
LENGTH: Length in bytes.
EVEN/ODD ADR: As described in the FTMP Software manual.
CACHE ADR: Rx.CACHE where x is 1, 3, or 4. This is the buffer that you reference in your task program to use task data block variables. It is usually at the bottom of the task's stack page. So as you increase the size of the Data block remember to modify the stack limit in the PSD.
NEXT PNTR: unused.

None of the present tasks on FTMP use the Data Control Block. Instead, all tasks run in PRIVILEGED mode and get data to/from system memory directly with WRT and RD commands (USER tasks are not allowed to access system memory with RD/WRT routines which is the reason for Data Blocks).

PSD block:

Implementation is pretty much as described in the FTMP Software manual. However, the PMR field warrants further discussion.
PMR -- Privileged/non-privileged status. *Zero* for non-privileged, *non-zero* for privileged. There is a significant difference between these two modes. In Privileged mode:

- You *can* use the bus service routines (RD, WRT, etc.) and thus can access system memory any time.

- All interrupts, except for timer, IPC and page faults are ignored. Thus you could overflow stacks or write to protected memory without interrupt. Of course, most interrupt handlers are not written so it really doesn't matter.

By contrast, in non-privileged or USER mode:

- You *cannot* use the bus service routine and thus have no direct access to system memory. Instead you must rely on Data Blocks to store things in system memory. If you try to use RD or WRT in USER mode, YOU WILL STALL THE SYSTEM (requiring system reboot) since you would invoke the unimplemented write protect violation

interrupt.

- Interrupts are not automatically masked in USER mode. In fact, many of the exceptions will crash the system since many of the interrupt handlers are unimplemented. Considering this risk, most users will inevitably run everything in privileged mode

Each time you add a task, one of each of the above blocks (Data Block excluded) must be created in TABLES.

A few final notes: Remember to re-assemble the TABLES file if you change it. Also, some of the fields can be changed dynamically using CTA. In particular, you could set the forward pointer in the Task Control Block to link in a new task dynamically. The Synthetic Workload program developed by CMU reconfigures the system this way. Every so often doing this might stall the system if you happen to set a pointer as the dispatcher is modifying it (this is especially true with R1 tasks). With respect to TABLES and the link file, system memory is divided into data and code. Code is generally being paged into local memory as processors need it while data is accessed with RD or WRT statements. Data can only be in pages 0-27 of system memory while pages 28-3F are code only. The boundaries are hard set in PROM in the page fault handling routine (PFAULT).

And last but not least: think carefully and keep straight when addresses are *local cache RAM* addresses and when they are *main memory* addresses. As a rule, and don't break this rule, main memory variables are defined in TABLES. So on the cross reference listing, any variable that is DEFINED IN tables is in main memory and must be referenced with RD/WRT. Otherwise it is a cache local.

1.5.4. Linking

Linker errors do not appear at the end of the job listing as they do for compiles and assembles. To see errors like undefined labels and such, go to the linklist file in the subdirectory LINKLIST.DIR. Load the linklist file into the editor and search for all occurrences of "ERROR". This linklist file also contains such valuable information as a memory map and a cross reference listing.

The link may be aborted if there is something like a syntax error in the link command file. In such a case, no linklist file will be produced. Instead, the word "ABEND", for "abnormal end", will appear in the first page of the job listing. An error code, which must be looked up in an appropriate manual [7], will be on the third or fourth page of the listing. Some of the most common reasons for a linker abort are:

1. one of the lines in the linker file doesn't end with a semicolon (;), and

2. there is a blank line (!!) in the link file (related to the first problem).

If the link is successful there will be a absolute load module in the **load** directory called *filename*.LOA.

1.6. Downloading

Now that you have compiled and linked a program task you want to run it on the FTMP. But before you can run the task on FTMP you have to get it there (i.e. move the absolute load module from the VAX to the FTMP). There are two steps to loading FTMP with your program. First, you must prepare the load module (*filename*.LOA) by stripping the PROM code and terminating blanks. There is a program developed by Ed Czeck the does this. To run it type RUN [EWC.BIN]FXLD.EXE. The "fixed" load module produced by this program is now ready to be loaded. The FTMP can be loaded by executing one of the following DCL command files (the [LALA.CAPS] directory is located on DISK$DEVPACK):

`[LALA.CAPS]GO.COM`	`Loads FTMP with its normal operating system (SCC) and applications under a three processor triad configuration.`
`[LALA.CAPS]FGO.COM`	`Loads FTMP with a fault injection version of the operating system (FSCC) and a three processor triad configuration.`
`[LALA.CAPS]2TRIADS.COM`	`Loads FTMP with its normal operating system and two processor triads.`
`[LALA.CAPS]1TRIAD.COM`	`Loads FTMP with its normal operating system and a single processor triad.`

If you want to load FTMP with your own memory image (load file) simply copy one of the above command files into your directory and change the line that loads the executive memory image to load your file instead (this statement is around line 90-100; use the EDT command FIND 'LOAD' to help find the line). Start the downloading process by giving the command @<command file> to the VMS prompt. For example, to load FTMP with a two processor triad configuration the command is

$ @2triads

Loading FTMP takes about 10 minutes.

The easiest way to tell if you loaded FTMP correctly is to see if the display terminal starts up. You can also use CTA to check main memory locations 6 00 (TIME_NOW) and 3 18 (TIME LO -- next R4 start time). If these do not change between checks the FTMP system is not running.

1.7. Debugging FTMP

Debugging a program running on FTMP is actually very difficult. The only way to check if your program is running is to have it write to select main memory locations and use CTA to check these locations. Also, FTMP is a real-time computer, so a (R4) program task will run 25 times a second.

However, when debugging, you will undoubtly want to run the task only once and check results. Therefore, provide another main memory variable that when set, runs the task once and resets the variable. Figure 1-4 is an example of a typical program in the debugging stage.

```
MODULE BEGIN
  <declarations...>
EXTERNAL MM.START, MM.CHKPT;      ... Main Memory Variables. //
INTEGER  MM.START, MM.CHKPT;
  <...>
DEFINE TASK TOBE
BEGIN
  INTEGER START;

  RD(MM.START,START,1);    ... START <-- MM.START            //
  IF (START EQL 1) THEN    ... Execute the following code    //
     BEGIN                 ... when START set to 1.          //
       <code...>
       WRT(MM.CHKPT,10,1);  ...  Mark that we got to this point. //
       <code...>
       WRT(MM.CHKPT,11,1);  ...  Second mark.  //
       <etc...>
       START = 0;
       WRT(MM.START,START,1);  ...  Only execute test code once. //
     END;
  RESUME(0);
END;
END FINI;
```

Figure 1-4: Program in Debugging Stage

There is actually a way to step through and breakpoint a program using the CAPS Test Adaptor. However, neither of the authors have used it for debugging. We refer the reader to the document *CAPS Test Adaptor User's Guide* [9] for information on breakpointing a program with the test adaptor.

1.8. Collecting Data

Once you have a working program task you will want to collect data from it. Typically this data will be in the form of timer values, although other forms like sensor values are feasible. In addition, as an experimenter you will want this data recorded in a file for later analysis. All of this is done with CTA and VAX command files. Data collection also involves tailoring your programs so that they dump an array of values into memory upon command (i.e. setting a memory value with CTA).

Figure 1-5 is an example of a typical command file for data collection. To run a command file type the following to the VAX prompt:

$ @<*command-file name*>

If you want the output saved in a file, use the /OUTPUT parameter on the command line. An good example of a more extensive data collection command file is the one to run the workload generator for FTMP implemented by CMU. This file is in [EFC.WORKLD]WRKLD.COM.

Data collection dumps are not very pleasant to look at (Figure 1-6), so the experimenter will probably

```
$ !  Comment line
$ !  This file collects 25 data sets.  For this particular
$ !  program task, FTMP dumps data into location F 00 upon
$ !  command (setting 5 D8 to -1).
$ COUNT = 0
$ LOOP:
$ ! Set FTMP memory location 5 D8 to -1 to start data collection.
$ MCR CTA
  MS 5 D8 = FFFF
  EXIT
$ ! Pause 2 seconds to let FTMP write out data.
$ WAIT 0:00:02
$ ! Look at the data.
$ MCR CTA
  ML 10 00 108
  EXIT
$ COUNT = COUNT + 1
$ IF COUNT .LT. 25 THEN GOTO LOOP
```

Figure 1-5: Data Collection Example

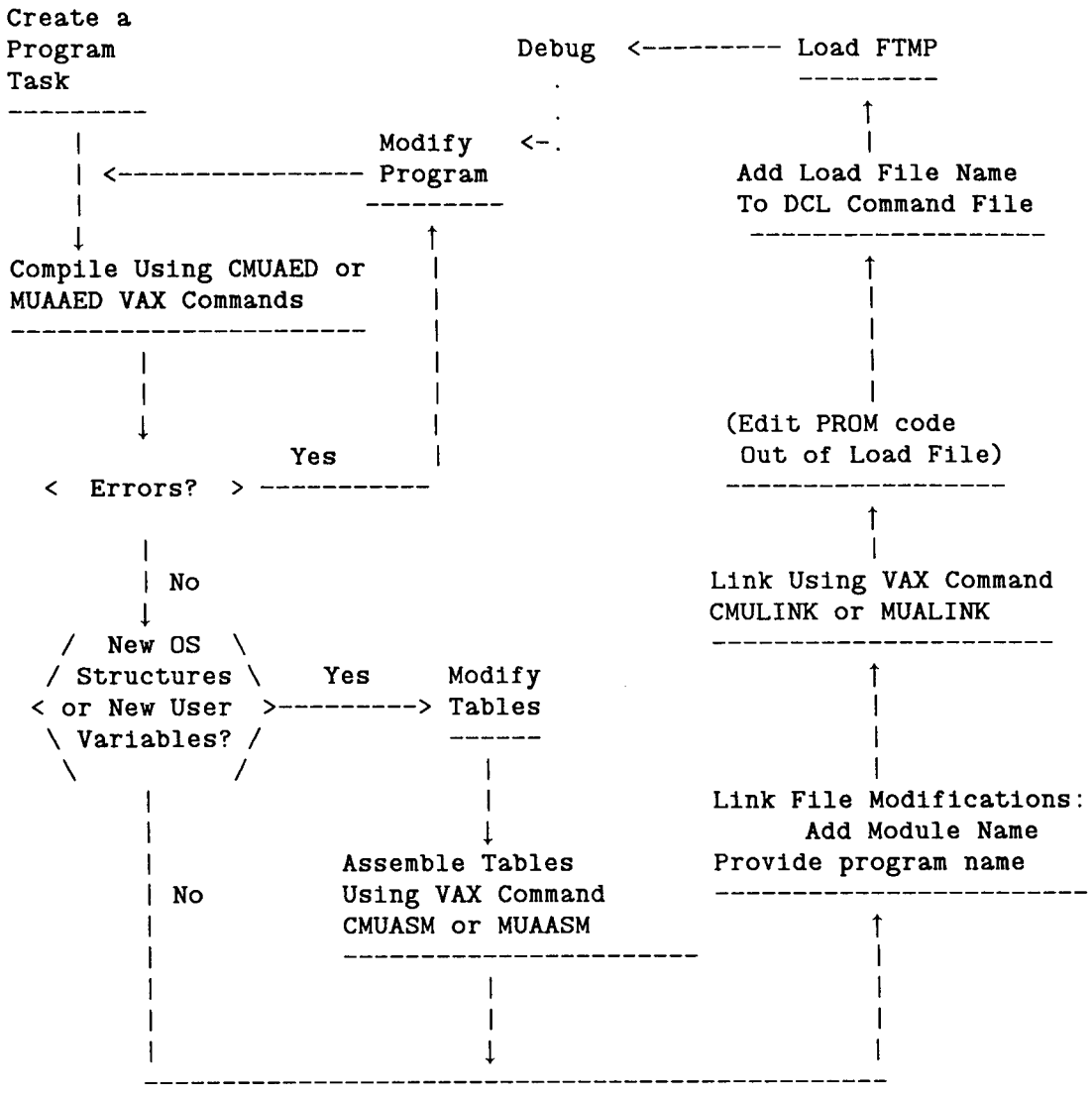
```
0032 34F4 0032 34F3 0032 34F2 0032 34F1 ! 0010 0000
0032 3591 0032 3591 0032 34F5 0032 34F4 ! 0010 0008
0032 3595 0032 3594 0032 3593 0032 3592 ! 0010 0010
0032 369E 0032 369D 0032 369C 0032 369C ! 0010 0018
0032 3785 0032 3784 0032 36A0 0032 369F ! 0010 0020
0032 3788 0032 3787 0032 3786 0032 3785 ! 0010 0028
0032 386B 0032 386A 0032 3869 0032 3868 ! 0010 0030
0032 3955 0032 3954 0032 386D 0032 386C ! 0010 0038
0032 3958 0032 3957 0032 3957 0032 3956 ! 0010 0040
0032 39D1 0032 39D0 0032 39CF 0032 39CE ! 0010 0048
0032 3A6E 0032 3A6D 0032 39D3 0032 39D2 ! 0010 0050
0032 3A71 0032 3A70 0032 3A70 0032 3A6F ! 0010 0058
0032 3B28 0032 3B27 0032 3B26 0032 3B25 ! 0010 0060
                     0032 3B29 0032 3B29 ! 0010 0068
```

Figure 1-6: Sample Data Collection Dump

want to write a program to decipher and analyze these dumps. An example of a 'C' program for analyzing timer value dumps is in [EFC.WORKLD]WRKLD.C.

1.9. Summary

Confused by all of the steps to preparing a program? The following flowchart of programming steps should help.


Create a Program Task
Modify Program
Compile Using CMUAED or MUAAED VAX Commands
Errors?
Yes
No
New OS Structures or New User Variables?
Yes
Modify Tables
Assemble Tables Using VAX Command CMUASM or MUAASM
No
Debug
Load FTMP
Add Load File Name To DCL Command File
(Edit PROM code Out of Load File)
Link Using VAX Command CMULINK or MUALINK
Link File Modifications: Add Module Name Provide program name

2. FTMP Operation From the VAX

There are various support hardware and software for FTMP installed on AIRLAB VAX system 10. This chapter outlines the support environment and operations for installing and using the support software and hardware. Specific questions on hardware should be addressed to the current hardware maintainer of FTMP at AIRLAB. Questions on the support software should be addressed to the current software maintainer.

2.1. FTMP Overview

2.1.1. FTMP Hardware Overview

The FTMP is mounted in a rack and composed of 10 identical boxes, each a Line Replaceable Unit (LRU), which are connected through a system bus. Each LRU has 19 printed circuit boards that make up its processor with cache memory, system memory, MIL-STD-1553A I/O port, and bus interface. The transfer bus of each processor, which connects the processor with its cache memory, is accessible through the bottom connector in the front of the LRU.

The FTMP support equipment is mounted in a rack beside the FTMP and is composed of the Collins Test Adapter (CTA), seven 1553 communication boards, the fault injector, and the PROM programmer. They are all connected to the VAX's UNIBUS.

The CTA can load programs into the FTMP and display 4 locations in the processor's cache memory. The CTA is connected to the transfer bus through the bottom connector in the front of the LRU that is initially acting as master, which is usually LRU A. The top connector controls access of the LRU to the system bus during loading. There is a shorted plug that when plugged in, the LRU is enabled to all the busses. LRU A has a computer controlled version of this shorted plug which is used by DCL command files that load FTMP (i.e. SHORTED and OPENED command).

The 1553 communication boards provide the FTMP a status display on a RS232-Terminal and communication with programs in the VAX. These boards have a microcomputer that allows them to be microprogrammed to do 1553-RS232 or 1553-UNIBUS communication. Each board has a separate UNIBUS base address. The last two numbers of the UNIBUS address are used as a tag for each board. They have the following tag numbers: 00, 08, 10, 18, 20, 28, and 30. Only two of the boards are used at any one time, and they are connected to two I/O ports each.

The fault injector is used to inject hardware faults into the FTMP for experimentation. It is connected between the subject Integrated Circuit (IC) and its board.

The PROM programmer loads the PROM board of the FTMP processor after it has been erased with UV light. Consult the FTMP hardware report [10] for further information on the PROM programmer.

2.1.2. FTMP Software Overview

The programs to use the FTMP's support equipment are written in the assembly language for the PDP 11/60 and run in compatibility mode in the VAX. Their names are CTA for the test adapter, M15 for the communication boards, and FIS for the fault injector. They can be invoked with the DCL command MCR <Program_Name>. Their command languages are described in the facilities software section of the FTMP software report [11].

Each of these programs has an interface program that allows them to access the UNIBUS. The names of their interface programs are CTINTG, INTER, and FISIN respectively. They communicate with their interface programs using mailboxes. The interface programs are written in VAX assembly language and FORTRAN, and are activated as detached processes when the VAX reboots. These programs and the FTMP load files reside in the VAX disk labeled DISK$DEVPACK.

2.2. VAX Initialization

The VAX's startup procedure should create the detached processes CTINTG, INTER, and FISIN. There execution can be verified by the DCL command SHOW SYSTEM. The detached processes initially created have the UIC 175,10 and can only be used by users whose UIC group number is 175 (all FTMP users UIC's have 175 in the group field).

Two 1553 communication boards are loaded with there microcode at boot time and should be working all the time. Which boards are being used could change at any time. Therefore if they are not working it is best to notify AIRLAB personnel.

2.3. Turning the FTMP On

All the power FTMP requires is provided by turning on the switch labeled "POWER" in the top panel of the support rack beside the FTMP. Once this is done the amber light to the right of the switch should be on. FTMP requires 28 VDC for its LRU's and 400 HZ for its fans. The green and red lights on the left of the switch indicate whether the 400 HZ is in or out of phase respectively. For safety reasons, the 28 VDC will not be applied if the 400 HZ is not on and in phase.

2.4. Loading the FTMP

The FTMP load files are in the LOAD.DIR subdirectories and their file type is LOA. The major three load files are:

- EXEC -- The normal operating system and application tasks.
- FEXEC -- A version of EXEC with the FSCC system task for automatic fault injection.
- LRUDIAG -- A standalone diagnostic program for one LRU (described in section 3.4).

As covered in section 1.6, the FTMP can be loaded by executing a DCL command procedure. The command procedure DISK$DEVPACK:[LALA.CAPS]GO.COM will load the FTMP with its normal operating system and applications (EXEC.LOA). The command procedure DISK$DEVPACK:[LALA.CAPS]FGO.COM will load the FTMP with the fault injection version of the operating system and applications.

2.5. FTMP Status Display Operation

2.5.1. Turning the Display Station On

The display station is an HP terminal and it can be turned on by a switch on the left back corner. It must be set to 4800 baud, full duplex, no parity, and the REMOTE switch must be depressed. The switch box the terminal is connected to must be set to the position labeled FTMP.

2.5.2. Loading the 1553-RS232 Communication Board

The 1553-RS232 communication board that interfaces the FTMP with the display station is loaded with its program every time the VAX reboots. This board can be reloaded by executing the DCL command procedure DISK$DEVPACK:[LALA.CAPS]DISPLAYxx.COM (xx = tag number on the board). Which procedure you run depends on the current interface board. Look for a tag on the board to get the board number.

2.6. FTMP Fault Injector Operation

2.6.1. Loading the 1553-UNIBUS Communication Board

The 1553-UNIBUS communication board that interfaces the FTMP with the VAX UNIBUS to do Direct Memory Access (DMA) is loaded with its program every time the VAX reboots. This board can be reloaded by executing the DCL command procedure DISK$DEVPACK:[LALA.CAPS]FISxx.COM (xx = tag number on the board). Which procedure you run depends on the current interface board. Look for a

tag on the board to get the board number. Before reloading this communication board the FISIN process (section 2.2) **must** be executing or have been executing at least once since the VAX rebooted, because the first time it executes it allocates the VAX memory page used for DMA.

2.6.2. Connecting the Fault Injector

The fault injector can be connected to one or more boards of LRU 3. During the connection of the fault injector to a board, the board must be disconnected from the LRU. Boards can be connected and disconnected from a LRU with the FTMP power on, except for the RAM and BGU boards which have CMOS IC's. These boards must have sockets and are extended using one or more extender cards. The IC's to be connected to the fault injector must be taken out of the board and replaced by FET extenders. The pins of each FET extender must be connected to the pins in the socket that they are going to be mapped to in the FIS command procedure. The IC's must be placed on top of the FET extenders so the IC pins are connected to the corresponding IC socket holes.

2.6.3. Injecting Faults

Both power supplies in the fault injector hardware must be turned on to inject faults. All units in LRU 3 should be placed in an active triad. Use the status display to verify that they work properly with the fault injector connected. The FIS program must be activated using the DCL command MCR FIS. After the prompt FIS> appears commands can be given interactively but usually a FIS command procedure for the types of IC's to be tested is executed. The default file type of FIS command procedures is CMD. For additional information in how to use the fault injector consult the FTMP test and evaluation report [12].

3. Helpful Hints

By the time you get to this chapter you should have a pretty good idea how to work with FTMP and how its real-time environment works. Hopefully, you have also written and run a simple program like the one described in Appendix III. If so, you undoubtedly have encountered problems and even extreme frustration over the turnaround time when you first write an AED program task to actually running it on FTMP. This chapter will hopefully explain how to shortcut the downloading process and explain what to do when problems are encountered with any of the various systems. You will undoubtly come to this chapter more than any other chapter in the manual. Hopefully, you can find answers to your questions here.

3.1. Dial Up Lines

For programmers that are working remotely, there is a dial up line to system 10 and to AIRLAB's bridge system.

```
VAX system 10:
        865-4407
        Must be set to 1200 baud, Full Duplex.

AIRLAB's Bridge System:
        865-4406
        1200 baud, Full Duplex
        Works best with a DEC-compatible terminal.
```

3.2. When FTMP doesn't work

3.2.1. FTMP Will not Load

Sometimes you will execute a command file to boot FTMP and that file will not load. There are variety of reasons for this, all of which should be checked.

1. If you are loading FTMP and get a message

   CT&A0302W FILE NOT FOUND - RESPECIFY

   this means CTA cannot find the absolute load module (.CAPS file) for downloading. If this happens, first check that the CAPS file does exist. Then make sure the CAPS file you are loading has a variable length record format (do a "DIR /FULL" of the file). If it has a some other record format, CTA does not know it exists. The easiest way to correct the format is to load the LOA file into the editor and save the file -- EDIT gives files variable length record format when it saves.

2. If the load stalls, check that RUNCTINTG is loaded (look for it in the process list with a "SHOW SYSTEM" command). If RUNCTINTG is not loaded, give the command @[LALA.CAPS]RUNCTINTG to start this process (see section 3.2.3) and try loading again.

3. Check that the cable from the test adaptor to LRU A is secure and the shorted switch is computer controlled.

4. If nothing else works, cycle the power to FTMP and try loading again.

3.2.2. HP2648 display terminal

Occasionally you'll boot FTMP and the display terminal will not start up. If the terminal doesn't boot, first make sure the HP is at 4800 baud and set to monitor FTMP. Another reason for the display terminal not working is the interface board needs to be booted. To do this, run the command file [LALA.CAPS]DISPLAYxx.COM (xx = tag number on the board). Which one you run depends on the current interface board. The last two digits on a tag on the board corresponds to the command file number. Thus, to boot the xx928 interface board do the following:

```
$ @[LALA.CAPS]DISPLAY28
```

If the terminal still doesn't boot, check that FTMP is running by using CTA to look at one of FTMP's clocks at location 6 00 or location 3 18. If the value in either of these locations does not change between checks, FTMP is not running.

3.2.3. VAX/FTMP interface and Collins Test Adaptor

Communication and downloading to FTMP is via a test adaptor connected to a Line Replaceable Unit (LRU A). A program on the VAX, called CTA, communicates with the test adaptor so the FTMP user can examine/change memory and download core images. The background process **CTINTG** needs to be present for CTA to work. Use SHOW SYSTEM to see if it is present. Start up this process with the following command:

```
$ @[LALA.CAPS]RUNCTINTG
```

If CTA stalls while using it, it's probably because CTINTG crashed (the DECWriter paper terminal will spew out a message). In this case you also need to start up CTINTG with the above command file. If CTA keeps crashing, check that the test adaptor/LRU connection is secure. If CTA still will not start up, you probably need to cycle power and reboot FTMP.

Finally, if the test adaptor was used to program PROMS or there was a power cycle, you may need to set the addresses that the Collin's test adaptor monitors. The addresses are 2001, 2002, 2003, and 2004. Set these using the "Set Addr" key on the test adaptor [9].

3.3. Failing and Repairing Processors

The easiest way to repair or fail a processor while FTMP is running is to give appropriate commands on the HP display terminal (pages 191-201, FTMP Software manual [11]). However, at times the user may want to reconfigure the system from the VAX (remote access) or from within a task program. In such

cases, it is still easiest to use display routines. Specifically, use the display input buffers so that FTMP thinks it's getting HP terminal input. The input buffer is LINE.BUFF (located at page 6 offset 4B) and expects a one word character count followed by ascii input characters. Put HP terminal commands in this buffer to fail or repair a LRU. Remember that on FTMP, bytes are ordered right to left, so the Least Significant Byte of a word is the first byte. Set LINE.READY (location 6 15) to 1 when you want the DISPLAY routine to read the input line. The memory array TRIAD.ID.TABLE (location 0 50) contains the state of all LRU's on the system so you can see the present configuration (page 49 of FTMP software manual [11]).

As an example, suppose we wanted to fail processor 6 using CTA. We would have to give the following commands:

```
CTA> MS 6 4B = 3          3 input characters
CTA> MS 6 4C = 5046       ascii "FP" for fail processor
CTA> MS 6 4D = 36         ascii "6"
CTA> MS 6 15 = 1          Transmit the line
```

For this procedure to work, R1 task number 1 and 2, "DISPLAY" and "SCC", must be linked in to the task structure. Most users build command files to fail or repair specific processors (i.e. the command @FAIL3 will fail processor 3) so they don't have to type the above commands every time they want to reconfigure the FTMP from the VAX.

3.4. LRU Diagnostic Program

LRUDIAG is a standalone diagnostic program for testing one LRU. If you suspect flaky performance in a unit it might be helpful to run this program on the LRU in question. The following procedure is used for running this diagnostic program.

Before every new test run, cycle the power on FTMP. Then set the test adaptor display to locations 20A0, 20A1, 20A5 and 20A4. Now plug the extender cable (normally plugged into LRU A) into the bottom plug of the LRU you want to test, and put the special shorted plug adaptor into the top plug. Next go into CTA and give the command @LRUDIAG. The diagnostic program will print several messages and ask you two questions -- just follow instructions. Pull the shorted plug before continuing after the second time the diagnostic program halts. See chapter 5 of FTMP software manual [11] for future discussion of LRUDIAG.

3.5. More on CTA

There are a few CTA commands not listed on pages 173-175 of the FTMP Software Manual [11]. Some of these commands are:

LIST address num: This command lists to the terminal 'num' words of cache memory starting at the given address. This only lists contents of cache for the LRU that the test adaptor is connected to.

SET address num = data: This is used to set cache memory on the LRU that the test adaptor is attached to. 'num' is the number of words to set to 'data' starting at the given address.

RESET, EXIT, HALT, RUN: Commands mainly used in command files for loading FTMP. The user will probably never need these commands so I will forgo discussion of them.

Also, the MSET command has additional fields not mentioned in the manual. Following is the new definition of MSET:

MSET page offset num = ($data_1$,$data_2$,..$data_{num}$): This is used to set 'num' system memory locations to '$data_1$' through '$data_{num}$' starting at the address defined by 'page' and 'offset'. 'Num' defaults to 1. The parenthesis are not required if you are setting all memory locations to the same value. (i.e. MSET 0 0 256 = 0, *zeros* the first memory page).

RTC: Show the value of the *Real Time Clock*.

Appendix I
System Bus Service Routines -- Additions

Pages 158-162 of FTMP Software Manual [11] describes the bus service routines available to the FTMP user. However, this document fails to describe how you should declare these routines if you wish to use them, especially the parameter types each routine uses. Also, some of the routines do not exist under the name given in the manual. Following is a list of the bus service routines and how each routine should be declared (As a reminder: **A** stands for ANY type of reference parameter, and **II** means INTEGER value parameter). See FTMP Software Manual [11] for a description of routines and arguments.

WRT(A,A,II); *instead of* **WRITE**.
RD(A,A,II); *instead of* **READ**.
HWRITE(A,A,I);
HREAD(A,A,A); If used to read RT.CLOCK, the first two arguments are longwords
NREAD(A,A,A);
NWRITE(A,A,I);
SREAD(A,A,I,II);
SWRITE(A,A,A);
SLREAD(A,A,A,II)
SNREAD(A,A,I,II)
READL(A,A,A);
WRITEL(A,A,A);
HOG.BUS();
RELEASE.BUS();

I will remind the FTMP user that these routines can only be called by a task executing in **privileged** mode.

Appendix II
Parameter Declarations

ARGUMENT TYPES

ABBREVIATION	SPELLING	MEANING
B	BOOLEAN	Reference
I	INTEGER	Reference
L	LONG	Reference
P	POINTER	Reference
R	REAL	Reference
IB	INPUT.BOOLEAN	Value
II	INPUT.INTEGER	Value
IL	INPUT.LONG	Value
IP	INPUT.POINTER	Value
IR	INPUT.REAL	Value
PR	PROCEDURE	Reference
BPR	BOOLEAN PROCEDURE	Reference
IPR	INTEGER PROCEDURE	Reference
LPR	LONG PROCEDURE	Reference
PPR	POINTER PROCEDURE	Reference
RPR	REAL PROCEDURE	Reference
LB	LABEL	Reference
S	SWITCH	Reference
A	ANY	Reference
BC	BOOLEAN COMPONENT	Reference
IC	INTEGER COMPONENT	Reference
LC	LONG COMPONENT	Reference
PC	POINTER COMPONENT	Reference
RC	REAL COMPONENT	Reference
BRCPR	BOOLEAN RECURSIVE PROCEDURE	Reference
IRCPR	INTEGER RECURSIVE PROCEDURE	Reference
LRCPR	LONG RECURSIVE PROCEDURE	Reference
PRCPR	POINTER RECURSIVE PROCEDURE	Reference
RRCPR	REAL RECURSIVE PROCEDURE	Reference
RCPR	RECURSIVE PROCEDURE	Reference

NASA
FORMAL
REPORT

FFNo 665 Aug 65

Appendix III
Program Example

Following is an example of how to create a program task on FTMP starting with compiling an AED program, to downloading and running the absolute object module on FTMP. For this example, the files are in the CMU directory and the file names will be EXAMPLE. Also, throughout this example, user response is underlined while italicized phrases are guiding comments.

Look at an AED program created earlier

```
$ TYPE [CMU.AED]EXAMPLE.AED
EXAMPLE BEGIN
  SYNONYMS HALT=RESUME;

  PROCEDURE RD(A,A,II), WRT(A,A,II);

  ... Main Memory Variables. //
  EXTERNAL EXAMP.EXEC1, EXAMP.EXEC2;
  INTEGER  EXAMP.EXEC1, EXAMP.EXEC2;

  DEFINE PROCEDURE HELLO.WORLD TOBE
    ... WRITES EXEC1 + 1 INTO EXEC2 //
    BEGIN
      OWN INTEGER TMP;    ... STACK LOCAL //
          INTEGER I;      ... NON-STACK LOCAL //

                          ... TMP <-- EXAMP.EXEC2 //
      RD(EXAMP.EXEC1, TMP, 1);
      I = 1;
      TMP = TMP + I;
                          ... EXAMP.EXEC2 <-- TMP //
      WRT(EXAMP.EXEC2, TMP, 1);

   RESUME(0);
    END;   ... PROCEDURE HELLO.WORLD //
END FINI;
```

Submit the above program for compile

```
$ CMUAED [CMU.AED]EXAMPLE
```

You will be sent mail when the compile finishes

Check if link file exists...

```
$ DIR [CMU.LINK]
 FRANK25    FRANK3     INTDIAG    IOLOOP     LRUDIAG    LUTEST     MEMQUES    NIOECHO
 OPCODE     PFTEST     PLLTEST    READLOOP   RMUXTEST   RVOTER     SCOOP      SEND
 SSAC       STAT       SYNC       SYNCPROM   SYNCS      SYNC1      TAPESYS    TASKR1
 TESTHOG    TESTLOOP   TESTPCL    TESTPF     TESTV      TESTVV     TEST1553   TIMERTST
 TMRTEST    TRIAD1     TRIAD2     TRIAD3     TSTLNK     TTY        TWOREADL   WRKLD
```

```
WRTTES     XFEXEC
```

It doesn't -- Create link file from prototype link file (EXEC)

```
$ EDIT [CMU.LINK]EXEC.LIN
         SEG     PROM            ;
*TYPE 'SCC'
00004600  IN      LIB(SCC)        ;
*TYPE 'TASK.R41'
00007900 TASK.R41   EQU AUTO.LAND ; TASK R41 IS "AUTO.LAND"
*TYPE 'TASK.R43'
         TASK.R43   EQU DUMMY    ;  PROCEDURE NAME
```

Set to our procedure name (HELLO.WORLD)

```
*SUB/DUMMY/HELLO.WORLD/
         TASK.R43   EQU HELLO.WORLD    ;  PROCEDURE NAME
1 substitutions
*TYPE -1
00008900                           ;
*TYPE -1
00008800  IN      LIB(TASKR43)     ; TASK R43 IS DUMMY TASK
```

Have Linker include our Relocatable Module

```
*SUB/TASKR43/EXAMPLE/
00008800  IN      LIB(EXAMPLE)     ; TASK R43 IS DUMMY TASK
1 substitutions
*TYPE +3
00009000 R43.STKLM  EQU *+6,0      ;
*TYPE -3:+1

00008800  IN      LIB(EXAMPLE)     ; TASK R43 IS DUMMY TASK
00008900                           ;
         TASK.R43   EQU HELLO.WORLD    ;  PROCEDURE NAME
00009000 R43.STKLM  EQU *+6,0      ;
00009100                           ;
*TYPE END-3
00010400  IN      LIB(TABLES)      ; SYSTEM MEMORY TABLES FULLY UPDATED
```

Save in New file

```
*EXIT [CMU.LINK]EXAMPLE.LIN
[CMU.LINK]EXAMPLE.LIN
```

Modify OS tables and add user variables to Main Memory by editing "TABLES"

```
$ EDIT [CMU.ASM]TABLES.ASM
00000100                                ; PDP1160.FTMP.ASM(TABLES)  9 JAN 82
*TYPE END-7:END
00133800                    VALUE  "E210"     ; PROM, TRIAD 3 ,1E00
00133900                    VALUE  "0522"     ;  T  , TRIAD 1 ,HIGH ORDER
00134000                    VALUE  "0532"     ;  T  , TRIAD 2 ,HIGH ORDER
```

```
00134100                    VALUE  "0502"     ;  T   , TRIAD 3 ,HIGH ORDER
00134200                    VALUE  "0602"     ;  C   , ANY TRIAD, 2ND HIGH
00134300                    VALUE  "0613"     ;  C   , ANY TRIAD, HIGH ORDER
00134400                    FINI              ;
```

Add New variables to the end (page F)

```
*INSERT
                                  EJECT          ;
                          END.OF.PAGEF  EQU   *          ;
                                  RES     "F00"-END.OF.PAGEF   ;
                          EXAMPL.EXEC1  VALUE     0    ;   VARIABLES ON PAGE F
                          EXAMPL.EXEC2  VALUE     0    ;
```

```
*TYPE END-9:END
00134100                    VALUE  "0502"     ;  T   , TRIAD 3 ,HIGH ORDER
00134200                    VALUE  "0602"     ;  C   , ANY TRIAD, 2ND HIGH
00134300                    VALUE  "0613"     ;  C   , ANY TRIAD, HIGH ORDER
                          EJECT          ;
                 END.OF.PAGEF  EQU   *          ;
                          RES     "F00"-END.OF.PAGEF   ;
                 EXAMP.EXEC1   VALUE     0    ;   VARIABLES ON PAGE F
                 EXAMPL.EXEC2  VALUE     0    ;
00134400                    FINI              ;
*TYPE 1
00000100                                 ; PDP1160.FTMP.ASM(TABLES)  9 JAN 82
```

Create ENTRY statements at the top of the file to make the new variables available externally.

```
*INSERT
                           ENTRY EXAMPL.EXEC2      ;
                           ENTRY EXAMPL.EXEC1      ;
```

Find OS task tables and check that TASK.R43 is linked in and the PSD for the new task is set up correctly.

```
*TYPE 'R4.LIST'
00063300                    VALUE  R4.LIST   ; TOP POINTER
*TYPE 'R4.LIST'
00070900 R4.LIST            EQU    *         ;
*TYPE -2:+30
00070700                    RES    104       ; SPACE FOR MORE TASKS
00070800                                     ;
00070900 R4.LIST            EQU    *         ;
00071000 R4.TASK1           EQU    *         ; TASK1 CONTROL BLOCK "AUTOLAND"
00071100                    VALUE  R4.TASK2  ; FORWARD POINTER
00071200                    VALUE  0         ; NOT USED
00071300                    VALUE  24        ; TIME LIMIT (6 MSEC)
00071400                    VALUE  0         ; FRAME COUNT
00071500                    VALUE  R41.DATA  ; NULL POINTER
00071600                    VALUE  R41.PSD   ; PSD POINTER
00071700                    VALUE  0         ; CONSTRAINTS
```

```
00071800                    VALUE  1          ; BIT NO
00071900                                       ;
00072000 R4.TASK2           EQU    *          ; TASK2 CONTROL BLOCK      "CWS"
00072100                    VALUE R4.TASK3     ; FORWARD POINTER ok
00072200                    VALUE  0          ; NOT USED
00072300                    VALUE  80         ; TIME LIMIT (20 MSEC)
00072400                    VALUE  0          ; FRAME COUNT
00072500                    VALUE  R42.DATA        ; DATA POINTER
00072600                    VALUE  R42.PSD    ; PSD POINTER
00072700                    VALUE  0          ; CONSTRAINTS
00072800                    VALUE  2          ; BIT NO
00072900                                       ;
00073000 R4.TASK3           EQU    *          ; TASK3 CONTROL BLOCK    "DUMMY"
00073100                    VALUE  0          ; FORWARD POINTER
00073200                    VALUE  0          ; (NOT IN TASK CHAIN)
00073300                    VALUE  24         ; TIME LIMIT (6 MSEC)
00073400                    VALUE  0          ; FRAME COUNT
00073500                    VALUE  R43.DATA   ; DATA POINTER
00073600                    VALUE  R43.PSD    ; PSD POINTER
00073700                    VALUE  0          ; CONSTRAINTS
00073800                    VALUE  3          ; BIT NO
00073900                    RES    104        ; SPACE FOR MORE TASKS
*TYPE 'R43.PSD'
00073600                    VALUE  R43.PSD    ; PSD POINTER
*TYPE 'R43.PSD'
00083800 R43.PSD        EQU    *              ;
*TYPE -1:+10
00083700                                     ; TASK PSD TABLES
00083800 R43.PSD      EQU    *               ;
00083900              VALUE R4.STACK         ; TOS
00084000              VALUE R43.STKLM        ; SKLM
00084100              VALUE TASK.R43,B       ; SPCR
00084200              VALUE 0                ; LENV
11184310              VALUE 1                ; PMR (ON)      PRIV. mode - ok
00084400              VALUE 1                  ; MAPPER (ON)
00084500              VALUE "F800"           ; ALL RUPTS (EXCEPT OVERFLOW) ENABLED
00084600              VALUE 0                ; PSD POINTER
00084700                                     ;
00084800              EJECT                  ;
*EXIT
[CMU.ASM]TABLES.ASM
```

Now assemble TABLES

```
$ CMUASM [CMU.ASM]TABLES
```

Wait for compile and assemble to finish

Submit a link

```
$ CMULINK [CMU.LINK]EXAMPLE
```

Check for "ABEND" in the results

```
1                        J E S 2  J O B  L O G  --  S Y S T E M  0 4 8 2  --  N
 16.19.04 JOB  410  IEF677I WARNING MESSAGE(S) FOR JOB CMULINK  ISSUED
 16.19.04 JOB  410  $HASP373 CMULINK  STARTED - INIT I2 - CLASS C - SYS 0482
 16.19.04 JOB  410  IEF403I CMULINK - STARTED - TIME=16.19.04
 16.20.26 JOB  410  CCI001C LINK    /00010.55/00136/00256/002037/0/4/0000/FM11P99
 16.20.26 JOB  410  IEC130I PARM     DD STATEMENT MISSING
 16.20.26 JOB  410  IEC130I MESSAGE  DD STATEMENT MISSING
 16.20.30 JOB  410  CCI001C COPYDATA/00000.34/00044/00256/000773/0/2/0000/FM11P99
 16.20.39 JOB  410  CCI001C PRINTALL/00000.38/00008/00256/000504/0/1/0000/FM11P99
 16.23.20 JOB  410  CCI001C ZLOAD   /00055.56/00060/00256/000273/0/3/0000/FM11P99
 16.23.20 JOB  410  IEF404I CMULINK - ENDED - TIME=16.23.20
 16.23.20 JOB  410  $HASP395 CMULINK  ENDED
0------ JES2 JOB STATISTICS ------
- 17 OCT 84 JOB EXECUTION DATE
-         46 CARDS READ
-      1,281 SYSOUT PRINT RECORDS
-          0 SYSOUT PUNCH RECORDS
-       4.27 MINUTES EXECUTION TIME
UQS0009 - END OF DATA SET.
    1      //CMULINK  JOB (FM11,P996),'M. REILLY',CLASS=C,REGION=256K,
  JOB  410
           // TIME=0010
           ***UQ       ACCOUNT FM11
           ***OUTPUT   OUTPUT WILL BE HELD - DO NOT RELEASE
           //XLINK     PROC MEMB=
           //LINK      EXEC PGM=LINK,PARM='XREF '
           //STEPLIB DD DSN=PDP1160.AEDCAPX.LOAD,DISP=SHR,UNIT=3350,VOL=SER=335018
           //SYSPRINT DD  DSN=&&TMPLIST,DISP=(,PASS),UNIT=SYSDA,SPACE=(CYL,(1,1)),
           //               DCB=(RECFM=FBA,LRECL=133,BLKSIZE=133)
           //SOURCE   DD  DSN=LFMTN.CMU.LINK(&MEMB),DISP=SHR,UNIT=3350,
           //               VOL=SER=335018
           //LIB      DD  DSN=LFMTN.CMU.COBJ,DISP=SHR,UNIT=3350,
           //               VOL=SER=335018
           //IM$FILE  DD  UNIT=SYSDA,SPACE=(CYL,(2,2)),
```

Link didn't abort -- Now check for linker errors in LINKLIST file.

```
$ EDIT [CMU.LINKLIST]EXAMPLE.LIS
1                        J E S 2  J O B  L O G  --  S Y S T E M  0 4 8 2  --  N
*TYPE ALL 'ERROR'
           19*  IN      LIB(CLRALLEL)    ; CLEAR ALL ERROR LATCHES
        ** ERROR DETECTED IN LINE NO. 106 . SYMBOL EXAMP.EXEC2 IS UNDEFINED
        ** ERROR DETECTED IN LINE NO. 106 . SYMBOL EXAMP.EXEC1 IS UNDEFINED
        NUMBER OF ERRORS DETECTED : 2
              FAIL.ERROR     DISPLAY       046FE 0          *NOT REFERENCED*
```

Two undefined symbols!!

```
*TYPE 107
           104*  ORG    0,0              ;
*TYPE -2:+4
           102* R33.STKLM  EQU *+6,0     ;
           103*                          ;
```

```
          104*  ORG     0,0                ;
          105*  IN       LIB(TABLES)       ; SYSTEM MEMORY TABLES FULLY UPDATED
          106*  END                        ;
        ** ERROR DETECTED IN LINE NO. 106 . SYMBOL EXAMP.EXEC2 IS UNDEFINED
        ** ERROR DETECTED IN LINE NO. 106 . SYMBOL EXAMP.EXEC1 IS UNDEFINED
*QUIT

$ TYPE [CMU.AED]EXAMPLE.AED
EXAMPLE BEGIN
  SYNONYMS HALT=RESUME;

  PROCEDURE RD(A,A,II), WRT(A,A,II);

  ...  Main Memory Variables.  //
  EXTERNAL EXAMP.EXEC1, EXAMP.EXEC2;
  INTEGER  EXAMP.EXEC1, EXAMP.EXEC2;

  DEFINE PROCEDURE HELLO.WORLD TOBE
    ...  WRITES EXEC1 + 1 INTO EXEC2 //
    BEGIN
      OWN INTEGER TMP;    ... STACK LOCAL //
          INTEGER I;      ... NON-STACK LOCAL //

                          ... TMP <-- EXAMP.EXEC2 //
      RD(EXAMP.EXEC1, TMP, 1);
      I = 1;
      TMP = TMP + I;
                          ... EXAMP.EXEC2 <-- TMP //
      WRT(EXAMP.EXEC2, TMP, 1);

   RESUME(0);
    END;    ... PROCEDURE HELLO.WORLD //
END FINI;
```

Reason for error: Main memory variables called EXAMP.EXEC in AED program, but called EXAMPL.EXEC* in TABLES -- change AED code.*

```
$ EDIT [CMU.AED]EXAMPLE.AED
EXAMPLE BEGIN
*TYPE 'EXAMP'
          EXTERNAL EXAMP.EXEC1, EXAMP.EXEC2;
*SUB/AMP/AMPL/
          EXTERNAL EXAMPL.EXEC1, EXAMPL.EXEC2;
2 substitutions
*TYPE +1
          INTEGER  EXAMP.EXEC1, EXAMP.EXEC2;
*SUB/AMP/AMPL/
          INTEGER  EXAMPL.EXEC1, EXAMPL.EXEC2;
2 substitutions
*TYPE 'EXAMP.'
                          ... TMP <-- EXAMP.EXEC2 //
*SUB/AMP/AMPL/
```

```
                              ... TMP <-- EXAMPL.EXEC2 //
1 substitutions
*TYPE 'EXAMP.'
                    RD(EXAMP.EXEC1, TMP, 1);
*SUB/AMP/AMPL/
                    RD(EXAMPL.EXEC1, TMP, 1);
1 substitutions
*TYPE 'EXAMP.'
                              ... EXAMP.EXEC2 <-- TMP //
*SUB/AMP/AMPL/
                              ... EXAMPL.EXEC2 <-- TMP //
1 substitutions
*TYPE 'EXAMP.'
                    WRT(EXAMP.EXEC2, TMP, 1);
*SUB/AMP/AMPL/
                    WRT(EXAMPL.EXEC2, TMP, 1);
1 substitutions
*TYPE 'EXAMP.'
String was not found
                    WRT(EXAMPL.EXEC2, TMP, 1);
*EXIT
[CMU.AED]EXAMPLE.AED
```

Recompile

```
$ CMUAED [CMU.AED]EXAMPLE
```

Wait for compile to finish

Relink

```
$ CMULINK [CMU.LINK]EXAMPLE
```

Modify DCL command file to load FTMP with EXAMPLE memory image.

```
$ EDIT 2TRIADS.COM
WRITE .      THIS PROGRAM STARTS UP 2 PROCESSOR AND MEMORY TRIADS.
*99
   99  LOAD EXEC.CAP
*SUB /EXEC.CAP/EXAMPLE.CAP/
   99  LOAD EXAMPLE.CAP
1 substitutions
*EXIT
SYS$DEVICE:[EFC.MANUAL]2TRIADS.COM
```

Wait for link to finish (you will get a mail message)

Check for "ABEND" in the results

```
1                      J E S 2  J O B  L O G  --  S Y S T E M  0 4 8 2  --  N
-
 16.27.43 JOB  425  IEF677I WARNING MESSAGE(S) FOR JOB CMULINK  ISSUED
 16.27.43 JOB  425  $HASP373 CMULINK  STARTED - INIT I2 - CLASS C - SYS 0482
```

```
 16.27.43 JOB  425  IEF403I CMULINK - STARTED - TIME=16.27.43
 16. 16.29.13 JOB  425  CCI001C LINK    /00010.89/00136/00256/002030/0/4/FM11P996
 16.29.13 JOB  425  IEC130I PARM     DD STATEMENT MISSING
 16.29.13 JOB  425  IEC130I MESSAGE  DD STATEMENT MISSING
 16.29.18 JOB  425  CCI001C COPYDATA/00000.36/00044/00256/000773/0/2/0000/FM11P99
 16.29.25 JOB  425  CCI001C PRINTALL/00000.37/00008/00256/000504/0/1/0000/FM11P99
 16.32.26 JOB  425  CCI001C ZLOAD   /00055.58/00060/00256/000276/0/3/0000/FM11P99
 16.32.26 JOB  425  IEF404I CMULINK - ENDED - TIME=16.32.26
 16.32.26 JOB  425  $HASP395 CMULINK  ENDED
0------ JES2 JOB STATISTICS ------
- 17 OCT 84 JOB EXECUTION DATE
-         46 CARDS READ
-      1,281 SYSOUT PRINT RECORDS
-          0 SYSOUT PUNCH RECORDS
-       4.72 MINUTES EXECUTION TIME
UQS0009 - END OF DATA SET.
    1      //CMULINK  JOB (FM11,P996),'M. REILLY',CLASS=C,REGION=256K,
  JOB  425
           // TIME=0010
           ***UQ       ACCOUNT FM11
           ***OUTPUT   OUTPUT WILL BE HELD - DO NOT RELEASE
           //XLINK     PROC MEMB=
           //LINK      EXEC PGM=LINK,PARM='XREF '
           //STEPLIB DD DSN=PDP1160.AEDCAPX.LOAD,DISP=SHR,UNIT=3350,VOL=SER=335018
           //SYSPRINT DD  DSN=&&TMPLIST,DISP=(,PASS),UNIT=SYSDA,SPACE=(CYL,(1,1)),
           //               DCB=(RECFM=FBA,LRECL=133,BLKSIZE=133)
           //SOURCE   DD  DSN=LFMTN.CMU.LINK(&MEMB),DISP=SHR,UNIT=3350,
           //               VOL=SER=335018
           //LIB      DD  DSN=LFMTN.CMU.COBJ,DISP=SHR,UNIT=3350,
           //               VOL=SER=335018
           //IM$FILE  DD  UNIT=SYSDA,SPACE=(CYL,(2,2)),
```

Linked ok

Load FTMP

```
$ @2TRIADS
Bit set
.      THIS PROGRAM STARTS UP 2 PROCESSOR AND MEMORY TRIADS.
.
.      MEMBERS OF TRIAD1 ARE LRU'S 0, 1 AND 2.
.
.      MEMBERS OF TRIAD2 ARE LRU'S 3, 4 AND 5.
.
.      THE MASTER IS LRU "A".
.
.      COOP.CAP LOADED IN MASTER
.
.      MASTER ISSUING BUS ENABLE/SELECT COMMANDS.
.
.    CLEARING SYSTEM MEMORY TO 0
.
.      BEGINNING LOAD OF EXEC MEMORY IMAGE
.
```

```
.     SYSTEM MEMORY LOAD COMPLETE
.
.     LRU'S 6,7,8,9,A,B ARE MARKED FAILED.
.
.     TRIAD.ID.TABLE, MRR.TABLE SHOULD BE ALTERED TO CHANGE
.     THIS CONFIGURATION.
.
.     SLOP IS SET TO 40 PER CENT OF R4 PERIOD.
.
.     STARTING 2 TRIADS
.
.     MASTER MAKING FINAL BUS ASSIGNMENTS
.
.
.    SYSTEM STARTED IN MULTIPROCESSOR MODE.
.
.    CONFIGURATION TABLES ARE LOCATED AS FOLLOWS:
.
.
.            TABLE              LOCATION  LENGTH
.
.   BUS INMUX SELECT CODE    0  20     12
.
.   C BUS ASSIGNMENTS        0  20     12
.
.   P, R AND T BUS ASSGN     0  38     12
.
.   MEMORY STATUS            0  44     12
.
.   PROCESSOR STATUS         0  50     12
.
.   ERROR LATCHES            1  00     48
.
. INITIATING TRANSFER OF CLOCK FROM MASTER
.
Bit is reset
.     DISCONNECTED FROM C BUS 1
.     DISCONNECTED FROM C BUS 2
.     DISCONNECTED FROM C BUS 3
.     DISCONNECTED FROM C BUS 4
.     DISCONNECTED FROM C BUS 5
.
$ MCR CTA
CT&A> ML 6 00 2              check that FTMP is running
                                        0000 0172 | 0006 0000
CT&A> ML 6 00 2
                                        0000 0176 | 0006 0000
CT&A> ML F 00 2
                                        0001 0000 | 000F 0000
CT&A> MS F 00 = 2
CT&A> ML F 00 2
                                        0003 0002 | 000F 0000
CT&A> EXIT                          It runs!
$
```

References

All references are available from AIRLAB personnel, NASA Langley Research Center.

[1] **UED Manual*
IBM, 1969.

[2] *AED Programmer's Guide*
Softech, 1973.

[3] *AED User's Guide*
Softech, 1973.

[4] *AED-CAPS Programmer Reference IBM 360/370 Version*
Rockwell Collins, 1979.

[5] *AEDCAPS Cross Compiler User's Guide*
Rockwell Collins, 1974.

[6] *CAPS Instruction Set Description*
Rockwell Collins, 1979.

[7] *CAPS Link Editor User's Guide*
Rockwell Collins, 1979.

[8] *CAPS Relocatable Cross Assembler User's Guide*
Rockwell Collins, 1976.

[9] *CAPS Test Adaptor User's Guide*
Rockwell Collins, 1979.

[10] *Development and Evaluation of a Fault-Tolerant Multiprocessor (FTMP) Computer, Vol I, FTMP Principles of Operations*
Charles Stark Draper Laboratory, 1983.
Contract Report (CR) 166071.

[11] *Development and Evaluation of a FTMP Computer, Vol II, FTMP Software*
Charles Stark Draper Laboratory, 1983.
CR166072.

[12] *Development and Evaluation of a FTMP Computer, Vol III, FTMP Test and Evaluation*
Charles Stark Draper Laboratory, 1983.
CR166073.

[13] *Introduction to AED Programming*
fourth edition, Softech, 1973.

[14] *User Utilities*
IBM, 1969.
Pages 63-119.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-89015		

4. Title and Subtitle	5. Report Date
FTMP Programmer's Manual	September 1986
	6. Performing Organization Code 505-66-21-02

7. Author(s)	8. Performing Organization Report No.
Frank E. Feather Carlos A. Liceaga Peter A. Padilla	
	10. Work Unit No.

9. Performing Organization Name and Address	11. Contract or Grant No.
NASA Langley Research Center Hampton, Virginia 23665-5225	
	13. Type of Report and Period Covered Technical Memorandum

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Washington, DC 20546-0001	

15. Supplementary Notes

Frank E. Feather is an employee of Carnegie-Mellon University.

16. Abstract

The Fault Tolerant Multiprocessor (FTMP) computer system was constructed using the Rockwell/Collins CAPS-6 processor. It is installed in the Avionics Integration Research Laboratory (AIRLAB) of NASA Langley Research Center. It is hosted by AIRLAB's System 10, a VAX 11/750, for the loading of programs and experimentation. The FTMP support software includes a cross compiler for a high level language called Automated Engineering Design (AED) System, an assembler for the CAPS-6 processor assembly language, and a linker. Access to this support software is through an automated remote access facility on the VAX which saves the user of the burden of learning how to use the IBM 4381.

This manual is a compilation of information about the FTMP support environment. It explains the FTMP software and support environment along many of the finer points of running programs on FTMP. This will be helpful to the researcher trying to run an experiment on FTMP and even to the person probing FTMP with fault injections. Much of the information in this manual can be found in other sources; we are only attempting to bring together the basic points in a single source. If the reader need any points clarified, there is a list of support documentation in the back of this manual.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Fault-Tolerance Multiprocessors AED Programming Fault Injection Data Collection	Unclassified--Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	43	A03