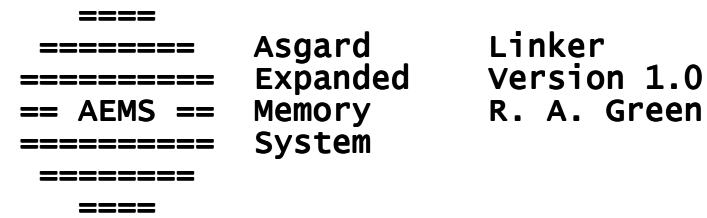
RAG SOFTWARE

AEMS PROGRAM LINKER V1



The AEMS LINKER is part of a series of programmer tools from RAG Software designed for the Asgard Expanded Memory System (AEMS). The AEMS provides a large paged memory for the TI 99/4A. The LINKER takes advantage of this large memory and provides significant facilities so that other programs and programmers can take advantage of the memory.

CONTENTS

INTRODUCTION

The AEMS LINKER is a tool for building assembler language program files from tagged object. It makes this process simple and straight forward. LINKER's main features are:

1. tagged object modules may be compressed and/or uncompressed,
2. tagged object modules may be absolute or relocatable and may contain complex relocatable values,
3. a library search can be done to resolve REFs,
4. a listing can be produced containing object information, memory maps and REF/DEF cross-references,
5. the program file can be built to load anywhere in the TI 99/4A's 64K address space and/or can be built to operate in overlay mode using the AEMS paged memory.

The AEMS LINKER is upward compatible with the RAG Software TI 99/4A LINKER. The LINKER requires the AEMS in order to run. The LINKER has four modes of operation depending upon the type of input in its control file.

1. Automatic. A single tagged object file is processed, a single library file is searched if necessary and a program file is produced. The program file will load as if the tagged object had been loaded by the E/A loader, excluding the E/A routines in low memory. That is, the object is loaded first in the 24K high memory block, then into the 8K low memory block.

2. Semi-automatic. A control file is processed which names one or more tagged object files to be processed and names one or more libraries to be searched. The resulting program file will load as if the tagged object had been loaded by the E/A loader, excluding the E/A routines in low memory. That is, the object is loaded first in the 24K high memory block, then into the 8K low memory block.

3. Complete Programmer Control. A control file is processed which names one or more tagged object files to be processed, names one or more libraries to be searched and which designates the memory layout for the resulting program file.

4. Overlay Mode. A control file is processed which names the tagged object files to be processed, names library routines to be processed and designates the memory layout and overlay structure for the resulting program file.

When operating in modes 1 to 3 LINKER can be directed to output "standard" E/A Option 5 programs, and is compatible with the TI 99/4A linker. LINKER normally outputs program files that must be loaded and executed using the AEMS Loader.

DISK CONTENTS

The distribution disk is a double sided single density disk containing the following files.

1ST/README	A few notes.
2/HISTORY	A short history of Linker releases.
ALNK1	The AEMS LINKER program.
ALNK2	The second segment of LINKER.
ALNKDOC	The documentation for LINKER to be printed by the TI WRITER formatter.
ALNKDOC1	Continuation of the documentation.
ALNKDOC2	Continuation of the documentation.
ALNKDOC3	Continuation of the documentation.
ALNKDOC4	Continuation of the documentation.
ALNKDOC5	Continuation of the documentation.
PRT10X	Patch file for Gemini 10X printer setup.
PRTDIST	Patch file for Generic printer setup (as distributed).
PRTNX1000	Patch file for Star NX1000 printer setup.
SCRDIST	Patch file for main screen default values (as distributed).
RAGLIB	A library of routines similar to those provided by the E/A, MM or XB for "standard" memory programs.
ZAEMSPAT/D	Documentation for ZAEMSPAT.
ZAEMSPAT	Public Domain program file patcher.

FAIRWARE

This package is being made available via the Fairware concept. If you like the package and are using it, send a donation to:

R. A. Green
1032 Chantenay Drive
Gloucester, Ont. Canada
K1C 2K9

And, at the same time, distribute complete copies of the package to your friends. If you have any suggestions for improvements or have found any bugs please forward them to the above address.

TAILORING THE LINKER

The LINKER is designed to work only on AEMS systems. It can be tailored to support various printers and to set the default processing options. Several patch files are provided on the disk which you can modify for your printer and main menu options. Two patch files, PRTDIST and SCRDIST will reset the linker to its original distributed state. Comments within the supplied patch files make modifying them for your setup easy.

You need read only enough of the PATCH program documentation to be able to run the patch program. Creating patches for a program is

fairly complex, but the patches here are already created by the author of the Linker, you need only modify them and run the patch program to apply them.

In order to produce a compact listing the printer should be set up to print in elite mode at 8 lines per inch. Most printers can do this although it is not required. To accomplish this printing, the LINKER will send a "setup sequence" of control characters to the printer before printing the first line of the listing, and will send a "reset sequence" of control characters to the printer after printing the last line of the listing. The LINKER will truncate any lines that are too long for the printer and will count the number of lines per page. The length of a line and the number of lines per page must be coordinated with the way your printer is set up.

The LINKER, as distributed, is set up for a generic printer and should work with any printer.

RUNNING THE AEMS LINKER

The RAG Software AEMS LINKER must be run using the AEMS Loader. The file name of the LINKER is ALNK1.

Linker Input Screen

The Linker screen is divided into three parts. The second part contains five input fields labeled: "Printer", "Library", "Options", "Control", and "Program". The Linker returns to the input fields after linking a program allowing you to do a batch of links at one time.

The "Printer" field specifies the name of the listing file to be used if any listing options are selected.

The "Library field specifies the name of the final library to be searched for unresolved REFs. A null entry means no library is to be searched. If there are no unresolved REFs at the end of a program, or if an overlay program is being linked, then this library is not used.

The "Options" field specifies the options to be used for this linker run. The options are specified as a sequence of one letter option codes. The codes may be entered in any order. The option codes are:

L - produce a listing of control statements and other optional data,
F - show full information in the listing about each tagged object module processed,
M - include in the listing a memory map of the memory image program,

X - include in the listing a cross-reference listing of all REFs and DEFs,
O - The program is being linked for overlay mode operation,
S - The output program file should be written as a standard E/A Option 5 program file.

The "Control" field specifies the name of your control file or of the single object file that you want processed.

The "Program" field specifies the name of the program file into which the Linker output is written. If required, more than one file may be written. The names for the second and following files are generated by incrementing the last byte of the given name.

The "Control" and "Program" fields are the only fields that must be specified.

Data Entry

During data entry the function keys perform as ordinarily defined by TI. In particular,

FCTN 1 (DEL)	Delete character,
FCTN 2 (INS)	Insert character,
FCTN 3 (ERASE)	Erase to end of field,
FCTN 4 (CLEAR)	Erase entire field,
FCTN 5 (BEGIN)	Begin execution of function,
FCTN 6 (PROCD)	Proceed with function,
FCTN 7 (AID)	File name selection from directory,
FCTN 8 (REDO)	Redo data in field,
FCTN 9 (BACK)	Terminate function,
FCTN = (QUIT)	Quit the Assembler,
ENTER	Move cursor to next field,
FCTN E (Up)	Move cursor to previous field,
FCTN X (Down)	Move cursor to next field,
FCTN S (Left)	Move cursor left,
FCTN D (Right)	Move cursor right.

When ENTER or FCTN X is pressed for the last field, the linking begins.

In the Directory Aid dialog box, the disk device name is entered in the usual way, including hard disk sub-directories, with or without the trailing period. The file name is selected by scrolling the cursor up and down and then pressing ENTER. The selected device and file name are placed in the input field. Pressing BACK cancels the directory aid without a selection.

The dialog box may display an error message. Pressing any key clears the message and returns to the input field.

Memory Usage

The linker uses AEMS memory pages, one at a time for two purposes. First, is the dictionary of REFs and DEFs. The dictionary maximum size is 24K. Second is the memory image of the program being linked. This memory image has a maximum size of 64K (the full addressing range of the 4A). During processing of an overlay program, overlay sections are written out as soon as they can be and the pages reused.

If the cross-reference option, "X" is given, the size required for the dictionary is increases considerably.

You may get three different messages from the LINKER about memory usage. "MEMORY FULL!" indicates that all the AEMS memory is used. "PROGRAM TOO LARGE" means that the program being linked is too large (check the memory blocks specified). "DICTIONARY FULL!" indicates that the dictionary is full, not specifying the "X" option may help this.

LINKER INPUT

The LINKER has three types of input file. First, a control file which is the only required input. The control file contains LINKER control statements and/or tagged object modules. The control file may be either VARIABLE or FIXED with a record length of 80. Tagged object modules in the control file begin with the tag "0" or tag >01 record and end with the colon record as is usual for object modules.

Second, tagged object files. A tagged object file is the normal output from an assembly. It ends with a record with a colon in column 1. Two or more tagged object modules can be combined into one file provided that all but the last colon record are deleted. Tagged object files must be FIXED 80. Tagged object files may have comment lines in them (which can be generated by the Assembler OBJREC assembler directive). These comment lines begin with an asterisk.

Third, library files. Library files are searched to resolve REFs. Only the modules required are processed. Each tagged object module in the library begins with one or more special header records and is terminated via the usual colon record. The header records are identified by a period in column 1. Beginning in column 2 of the header record is a list of the names that can be resolved by the module. The list contains 1 to 6 character names separated by commas with no intervening blanks (the first blank stops the scan of the header record). A name can not be started on one header record and continued onto the next. Library files are always FIXED 80.

The LINKER is a powerful tool so that describing its functions sometimes gets complicated. At the same time, LINKER is easy to

use if not all of its function is required for the job. If you do get bogged down in the following sections skip them and go directly to the examples at the end. The first three or four examples will show you how to do the easy things easily.

NOTATION

Throughout the remainder of this document, we will talk as though the LINKER were actually loading the object programs. The LINKER does not actually load the programs, it builds a program file that must be loaded by some other means (ie. the AEMS Loader, Option 5 of E/A, Option 3 of TI WRITER). Speaking of the LINKER in this way makes it easier to describe and to understand the functions of the various control statements.

Programs processed by the linker can be subdivided logically into sections and physically into file segments. The logical sections are mainly in overlay programs. There is the "root section" of the program which is not overlayed, and the "overlay sections" only one of which is mapped into the address space at any one time. Due to the size restrictions on PROGRAM files, each section of a program may be written as one or more file segments.

LINKER EXPRESSIONS

In the descriptions of the LINKER control statements, given below, the term "linker expression" will be used. A linker expression is a series of arithmetic operations on symbols and/or constants. The operations: + (addition), - (subtraction), * (multiplication) and / (division) are performed in a strict left to right sequence. A symbol used in an expression must be the linker origin symbol, "$", or must have been previously defined by appearing as a DEF in an object module or as the second operand of an ORIGIN or EQU statement. The value of a symbol is the address at which the symbol is loaded. Constants may be written as decimal numbers (138) as hexadecimal numbers (>8A) or as character strings ('HZ').

LINKER ORIGIN

The LINKER maintains an "origin" which represents the location in memory just past where the previous relocatable object module was loaded, or represents the value assigned by an ORIGIN or OVERLAY statement. The linker origin is always maintained on a word boundary (i.e. the address is always even). The symbol "$" represents this value in linker expressions. Loading absolute object code has no effect on the linker origin.

LINKER CONTROL STATEMENTS

LINKER control statements are used to control the various functions of the LINKER. They are entered into a control file via an editor (either the TI WRITER or E/A editor will do). Control statements are written beginning in column 1 as a control word followed by one or more operands and then optionally followed by comments. They are similar to an assembler language statement except that there is no label field. The operands are separated from the control word by one or more blanks. The operands are separated by commas. The comments are separated from the operand field by one or more blanks. In addition, comment lines which begin with an asterisk in column 1 may be entered. In addition to control statements, the control file may contain tagged object modules.

EXAMPLE

```
*        THIS IS A COMMENT LINE
LOAD DSK1.OBJ1           MAIN PROGRAM
LOAD DSK1.SUB1           SUBROUTINE
LOAD DSK1.SUB2           SUBROUTINE
| |      |                  |
| |      |                  Comment
| |     Operand
| Control Word
Column 1
```

Some LINKER control statements require a file name. These file names may be specified in any of three ways.

1. The full device/filename written in the usual TI fashion. Examples:

   ```
   DSK1.OBJ1
   DSK.diskname.OBJ1
   WDS1.directory.OBJ1
   ```

2. The disk number may be specified as an asterisk to indicate the same drive as specified for the control file. Example:

   ```
   DSK*.OBJ1
   ```

3. The entire device, device.diskname or device.directory may be replaced by an asterisk indicating the same device, device.diskname or device.directory as the control file. Example:

   ```
   *.OBJ1
   ```

Note that this same filename convention can be used on the Linker input screen. Only the "Control" file name must be specified in full.

The LOAD Statement

The LOAD control statement directs the LINKER to load tagged object modules. The LOAD statement has two formats. When loading an object file, the LOAD statement has a single operand, the name of the file to be loaded. When loading a routine from an object library the first operand is the library file name and the second is the name of the routine to be loaded. Using the LOAD statement to load library routines allows you to position the routines relative to other code as opposed to the automatic library search which usually positions routines at the end of a program. This positioning can be important in overlay programs.

A LOAD statement is equivalent to placing the actual tagged object modules into the control file.

Examples

```
LOAD DSK1.OBJECT1
LOAD DSK1.RAGLIB,VMBR  Load from library
LOAD DSK2.OBJECT2
LOAD DSK*.OBJECT3      SAME DISK AS CONTROL FILE
LOAD DSK.DISKNAME.OBJECT4
LOAD *.OBJECT5
```

The DEFs from the tagged object modules are entered into the LINKER's dictionary and may be used in expressions in following LINKER control statements.

If the "F" option was specified, LINKER will display information about the object modules loaded in the listing. The information consists of a line giving the type of object (compressed, uncompressed, relocatable or absolute) the size and address where a relocatable module is loaded, and the IDT data that was specified at assembly time. A line will be printed for each DEF in the module giving its loaded memory address; a line will be printed naming each REF in the module; and a line will be printed for each COMMON defined in the module giving its size. The colon record, which usually identifies the assembler which produced the object module and any comment lines within the object file will also be printed.

The LIBRARY Statement

The LIBRARY control statement directs the LINKER to search the named library file for any tagged object modules that contain DEFs for any &currently unresolved REFs. Note that only REFs that are currently unresolved will be searched for. If other tagged object modules are loaded after the LIBRARY statement has been processed that contain new REFs, these new REFs will remain unresolved unless they are resolved from additional tagged object modules or other LIBRARY searches. Usually then the LIBRARY statement will follow the LOAD statements in a control file.

Examples

```
LIBRARY DSK1.RAGLIB
LIBRARY DSK.DISKNAME.FILE
LIBRARY DSK*.LIB
LIBRARY *.LIBX
```

Note that one final library search may be done after the control file has been processed. The library searched is the one specified via the LINKER initial menu.

The ENTRY Statement

The ENTRY control statement is used to specify the entry point of the program file. This is the point at which execution begins when the program file is actually loaded. The ENTRY statement has a single operand, a linker expression that specifies the entry address.

The ENTRY statement is optional. The entry point for the program is determined by the LINKER as follows (in lowest to highest priority order):

1. entry at the first byte loaded,
2. entry as specified via the first tag 1 or tag 2 field processed (a tag 1 or tag 2 is generated when a symbol is named on the assembler END statement),
3. entry as specified via the ENTRY control statement.

Note that standard E/A Option 5 program files, by definition, begin execution at the first byte of the first segment. LINKER will insure that this is the case by producing 2 or more segments if necessary.

Examples

```
ENTRY SFIRST
ENTRY BEGIN
ENTRY BEGIN+>100
```

The BLOCK Statement

The BLOCK control statement is used to specify blocks of memory to be used for automatic loading of relocatable tagged object. (Absolute tagged object is, of course, loaded where it must be.) The BLOCK statement has three operands: the block number, the address of the beginning of the block and the size in bytes of the block. All three operands may be linker expressions (although they will usually be constants). The block address is adjusted upwards to a word boundary if necessary. The LINKER can handle four blocks of memory so that the block number (first operand) must be in the range 1 to 4.

The LINKER loads tagged object into the blocks of memory in a way analogous to the way the E/A or Mini Memory loads tagged object. It searches the memory blocks in order to find space to load the object modules. Ordinarily, the BLOCK statements will precede the LOAD statements in the LINKER control file. Note that if blocks are re-specified or if two BLOCK statements specify the same or overlapping memory blocks object modules could be loaded over top of previously loaded modules.

Examples

```
* Block definitions for non-overlay programs
BLOCK 1,>A000,>6000     (HIGH MEMORY)
BLOCK 2,>2000,>2000     (LOW MEMORY)
BLOCK 3,>6000,0         (CARTRIDGE RAM)
BLOCK 4,>4000,0         (DSR RAM)
*
* Block definitions for overlay programs
BLOCK 1,>2000,>2000
BLOCK 2,>A000,>6000
BLOCK 3,>6000,0
BLOCK 4,>4000,0
```

The above examples are the block definitions that the LINKER has when it begins executing depending upon whether or not the "O" option is specified. Note that blocks 3 and 4 have a length of zero so that nothing will be loaded in these blocks. In an overlay program the block definitions are only used during processing of the root section.

The EQU Statement

The EQU statement is used to define the value for a symbol. The EQU statement has two operands. The first is a linker expression that specifies the value to be assigned to the symbol that is the second operand.

EQU statements in the control file could be used to define symbols for REFs that would otherwise remain unresolved. For example, suppose you have a relocatable object program that has REFs to the standard routines VMBR and VMBW. Further suppose you want the program to execute in the Extended BASIC environment using XB's routines in low memory. The control file could be coded as shown below.

```
EQU >2024,VMBW          DEFINE XB'S VMBW
EQU >202C,VMBR          DEFINE XB'S VMBR
LOAD DSK1.OBJECT             THE PROGRAM
```

The FLAG Statement

The FLAG control statement is used to change the second byte of the program file flag word for standard E/A Option 5 program files. The value for the flag byte is the only operand of the statement. If no FLAG statement is used, the second flag byte value will be >FF for all but the last segment of a program, and will be >00 for the last segment. Section "Program Files" defines the use and various values for the flag word of program files. The main use for the FLAG statement is in linking GPL programs for loading into a GRAM device.

Examples

```
FLAG >09                MODULE FOR ROM BANK 1
FLAG >01                MODULE FOR GRAM 0
FLAG >47
```

The ORIGIN Statement

The ORIGIN control statement is used to specify the location in memory for the next relocatable tagged object module to be loaded. The ORIGIN statement cancels LINKER's automatic or semi-automatic mode. Once an ORIGIN statement has been used, the LINKER no longer searches the memory blocks for areas to load the object modules. All modules are simply loaded sequentially from the specified origin unless another ORIGIN or OVERLAY statement is encountered.

The ORIGIN statement has two operands. The first is a linker expression that specifies the memory location. This location is adjusted upwards to a word boundary if necessary. The second operand is optional, if specified it must be a 1 to 6 character name. This name is placed in the LINKER's dictionary just as DEFs from object modules are. The value assigned to the symbol is the value of the first operand. The symbol can be used to resolve REFs.

Examples

```
ORIGIN >A000.
ORIGIN >2000,LOW            DEFINE SYMBOL "LOW"
ORIGIN LOW+>1000            ORIGIN WILL BE >3000
```

It is important to remember that once an ORIGIN statement is used, relocatable object modules are loaded sequentially in the TI 99/4A 64K address space (wrapping from >FFFF to >0000 if necessary). As with the BLOCK statement, it is possible with the ORIGIN statement to cause LINKER to load one object module over the top of another. You are in complete control and must exercise that control. LINKER assumes you know what you are doing. An ORIGIN statement could precede each LOAD statement thus giving you complete control of where each module is loaded.

Absolute object modules or sections of modules which are absolute are not affected by ORIGIN statements. They are always loaded at the location specified in the object module.

ORIGIN statements should be used only in the root section of an overlay program. The OVERLAY statement specifies the origin for the overlay sections.

The OVERLAY Statement

The OVERLAY control statement is used to specify the beginning of a new overlay section of a program. The OVERLAY statement is only allowed if the "O" option was specified on the LINKER input screen. The OVERLAY statement has a single operand, the "overlay level". The overlay level is a number in the range 1 to 9 specifying the level in the overlay tree. A description of program overlay techniques and use of the OVERLAY statement is given later in section "OVERLAY PROGRAMS".

Due to hardware requirements of the memory card the origin of each overlay section must be on a 4K boundary. The OVERLAY statement will adjust to a 4K boundary as required.

When an OVERLAY statement is encountered, three processing steps are initiated.

First, the LINKER completes processing of the previous overlay (or root) section of the program. Any unresolved REFs in that section are assumed to be REFs to routines in the next higher level of overlay. For each of these unresolved REFs an overlay stub is built. The LIBRARY statement can precede the OVERLAY statement to force the LINKER to resolve REFS for the previous section. If the previous section was the root section then the "overlay manager" code is also built.

The second processing step is to allocate any COMMON sections defined in the previous section.

The third and final processing step is to adjust the linker origin. The origin is aligned to the next 4K boundary greater than or equal to the current linker origin excluding addresses >0000 to >1FFF and/or >4000->9FFF.

The names of the stubs built by the Linker are the same as the subroutine names except that they are in lower case.

Examples

```
OVERLAY 1
OVERLAY 2
```

The PATCH Statement

The PATCH control statement can be used to make patches to the loaded program. The PATCH statement has two operands. The first is the location to be patched specified as a linker expression. The second operand is the patch data. The patch data may be specified in one of three ways: as a ">" followed by a string of hexadecimal digits; as a character string in single quotes; or as a linker expression. All patching is done a word (or 2 bytes) at a time, if the specified patch data is less than a word the data is extended by adding zeros to a hexadecimal string and a byte of >00 to character strings. The linker expression is taken as a word value.

Examples

```
     PATCH PGM1+>A00,>FF037AFF         HEXADECIMAL PATCH
     PATCH >A010,'TEXT'                CHARACTER PATCH
     PATCH   16+PGM2,PGM1+50           EXPR - DATA PGM1+50
     *
     *      PATCH A BRANCH AT LOCATION >A000 TO
     *      ROUTINE "MAIN"
     PATCH >A000,>0460                 BRANCH INSTRUCTION
     PATCH >A002,MAIN                  ADDRESS OF ROUTINE "MAIN"
```

If the "F" option has been selected, LINKER will display the existing data in the listing before patching. It is possible to patch data into areas of memory that do not have any object code loaded previously.

The VERIFY Statement

The VERIFY control statement can be used to verify the contents of some loaded part of the program. It is often useful to verify data before patching it so that you are sure you are patching the correct location.

The VERIFY statement has two operands. The first is the location to be verified specified as a linker expression. The second operand is the verify data. The verify data may be specified in one of three ways: as a ">" followed by a string of hexadecimal digits; as a character string in single quotes; or as a linker expression. All verification is done a word (or 2 bytes) at a time, if the specified verify data is less than a word the data is extended by adding zeros to a hexadecimal string and a byte of >00 to character strings. The linker expression is taken as a word value.

If the data in memory is not the same as that specified, an error is caused. If the "F" option has been selected, LINKER will display the data in memory.

Examples

```
VERIFY PGM+>100,>0A1B2C3D0000
VERIFY >20A8,'TEXT'
VERIFY PGM1+50,PGM2
```

The COMMON Statement

The COMMON control statement can be used to position a COMMON section within the program and can be used to specify a size for the COMMON section. The statement has two operands, the first, the name of the COMMON section; the second the optional length of the section.

If the COMMON statement is not used, any defined COMMON sections will be placed at the end of the various program sections. If a size is not specified for a COMMON section then the maximum size specified in the already loaded object will be used.

If an object file contains a definition of the COMMON name as a regular DEF (that is, assembled data or code is given the name of the COMMON) then that definition will override the COMMON definition and its length. This means that a COMMON can be defined in one assembly and then data assembled "into" it in another assembly.

Examples

```
COMMON WORK             Position "WORK" here
COMMON TABLE,1024       Position and set size
```

OVERLAY PROGRAMS

The AEMS provides you with a large memory. The 9900 microprocessor in the TI 99/4A, however, is limited to a 16 bit address or a 64K "address space". Further, half of this 64K address space is dedicated to special purpose code -- DSR ROM, OS ROM, Cartridge ROM/RAM, etc.

To provide access to the large memory, AEMS "maps" 4K "pages" into the microprocessor's address space at addresses >2000, >3000, >A000, >B000, >C000, >D000, >E000 and >F000. This gives access to 8 different 4K pages of RAM at any one time in addition to the 8K of unmapped RAM that may be in a cartridge at address >6000 or any DSR RAM at address >4000.

While programming in a mapped memory environment is not as easy as programming in a large linear address space, it is very nice to have large quantities of RAM available directly to the microprocessor. In programs there are two almost independent uses of memory. The first use is for the executable code of a program and the second is for the data on which a program works.

The AEMS LINKER with its "overlay" processing solves some of the large code problem. The library routines provided with the AEMS solve some of the large data problem. Both the LINKER and the library routines provide the programmer access to the large memory without having to know the details of how the memory mapping works and without having to write extra code. (Section 'SYSTEM INFORMATION' gives the details for those who want them).

Writing any large program requires some discipline on the programmer's part. This discipline is even more important if the program is to execute in a small address space via a mapping mechanism. It is always good practice to subdivide a large program into relatively small subroutines. It is also always good practice to make these subroutines "independent". An independent subroutine is one that depends only on clearly defined data: data passed as parameters, data in a defined "COMMON" area, or data defined and contained within itself.

If the code in a large program consists of a main program and several independent subroutines with a "calling pattern" that can be diagrammed as a "tree" then that program can be segmented into an "overlay structure" by the LINKER. All code to perform the overlay (or mapping in and out of pages) can be added automatically by the LINKER without the programmer having to explicitly write code to perform the function. The programmer's task is simply to write his program as a series of "small" independent subroutines, which is good programming practice in any event.

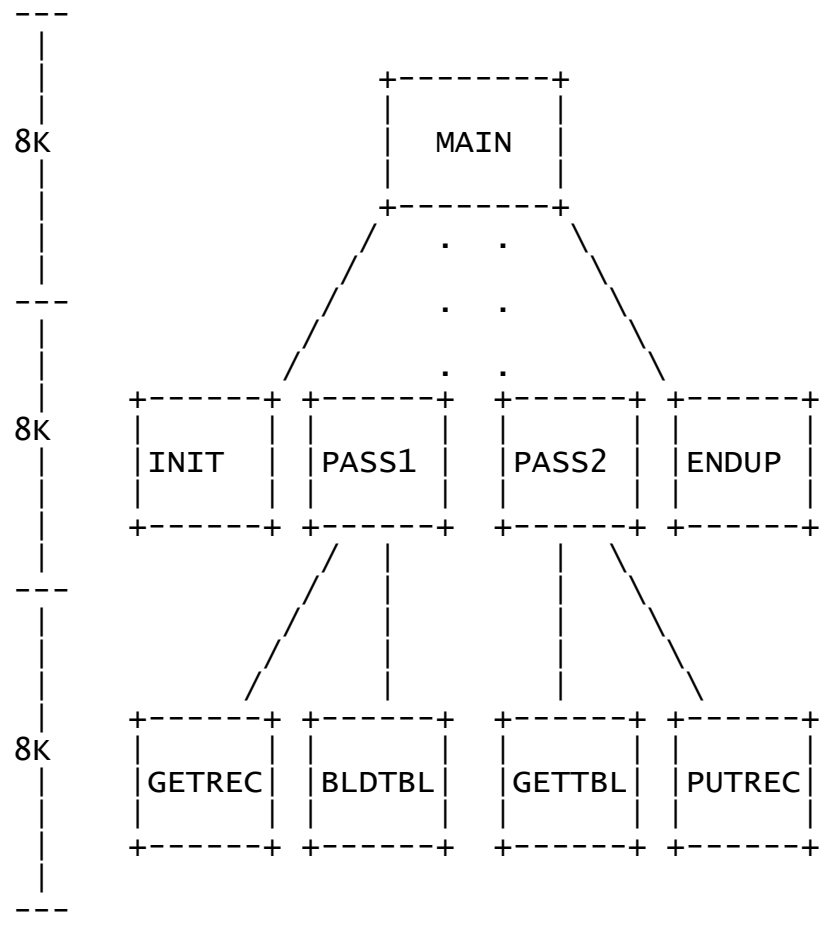
In order to demonstrate what we are talking about here we will use an example. Suppose we have a well designed and written "large" program. The source for the program is shown below, written in a "pseudo" language.

```
* Main Program
    CALL INIT
    CALL PASS1
    CALL PASS2
    CALL ENDUP
    END

* Initialization Subroutine
    ENTRY INIT
    SETUP VDP
    SETUP TABLES
    GET USER INPUTS
    OPEN FILES
    RETURN
    END

* Pass 1 Processing
    ENTRY PASS1
    CALL GETREC
    CALL BLDTBL
```

```
      LOOP TILL EOF
      RETURN
      END

* Pass 2 Processing
      ENTRY PASS2
      CALL GETTBL
      CALL PUTREC
      LOOP TILL END OF TABLE
      RETURN
      END

* End of Program Processing
      ENTRY ENDUP
      CLOSE FILES
      ISSUE MESSAGES
      RETURN
      END

* Get Input Records
      ENTRY GETREC
      GET RECORD
      CHECK RECORD
      RETURN
      END

* Output Record
      ENTRY PUTREC
      BUILD OUTPUT RECORD
      PUT RECORD
      RETURN
      END.

* Build Data Tables
      ENTRY BLDTBL
      ENTER DATA FROM RECORD
      RETURN
      END

* Lookup Table Entry
      ENTRY GETTBL
      CALCULATE TABLE POSITION
      PICK OUT VALUES
      RETURN
      END
```

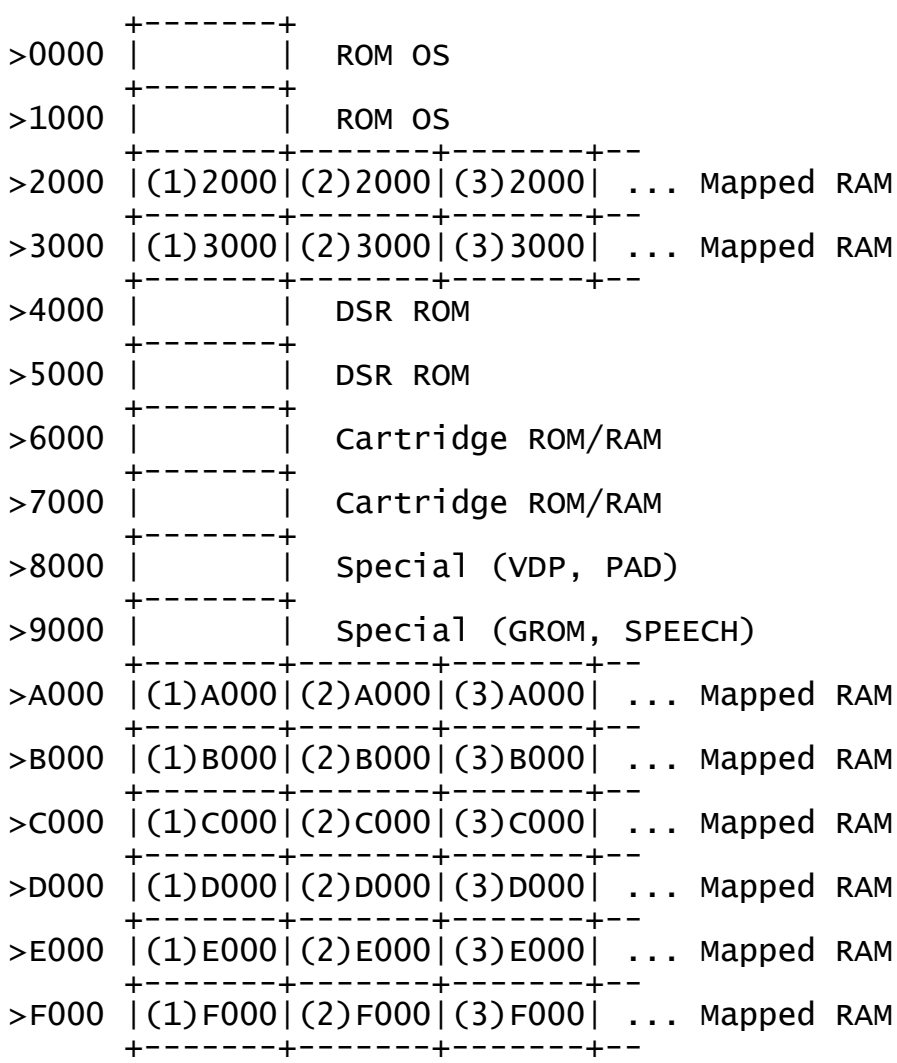
In this program we have a main program and 8 subroutines. This is a "large" program, so let's assume each of these routines is just under 8K bytes. Then our large program is 9 x 8K = 72K bytes, which is indeed a large program for the TI 99/4A with 32K RAM.

Now, let's view the "calling tree" of this program. It is shown below.



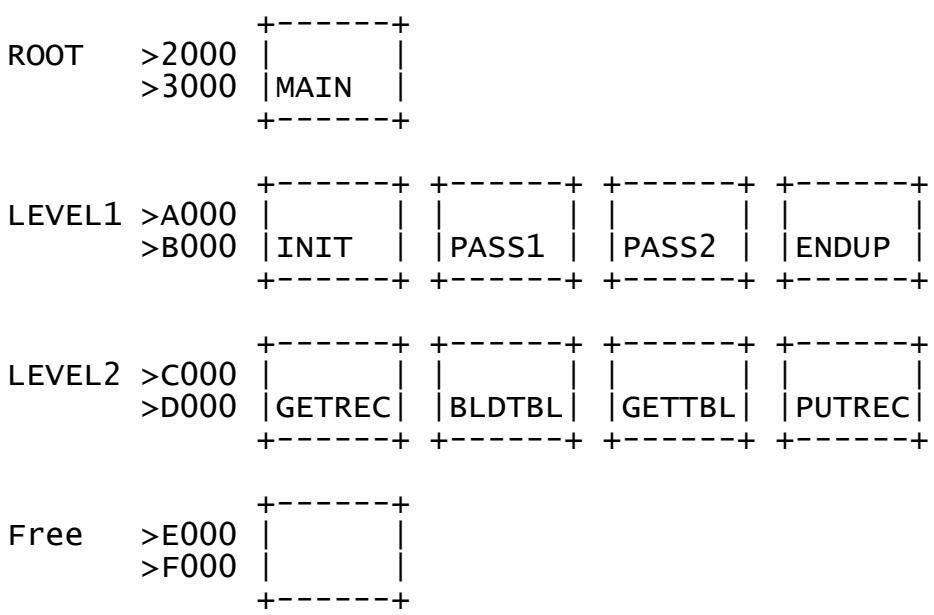
You can see from the diagram that, while the total program size is 72K, the depth of the calling tree is only 24K. The depth of the tree (or addressing space required) is the important measurement in a mapped memory environment.

Now, let's look at the addressing space and RAM availability of the TI 99/4A with AEMS.



This diagram shows the fixed area of the TI 99/4A addressing space and shows the mapped area with each 4K having the possibility of being mapped to several different pages of extended RAM.

Now we can see that it is possible to map our "large" program into the AEMS as shown below.



This mapping, of course, assumes that the CALLs somehow map in the correct pages at the correct addresses. The primary function of the overlay processing in the LINKER is to supply the code necessary to do the mapping at each CALL. It does this by building a small stub for each called routine in the caller's section plus a small overlay manager routine in the root section.

This overlay technique can be a powerful tool. In our example we have a 72K program and yet still have 8K of address space free (>E000 to >FFFF). he LINKER control statements necessary to produce the overlay program for the above example are shown in Examples 9 and 10 later in this manual.

To summarize overlaying:

1. The program must be written as small independent relocatable subroutines.
2. All CALLs must be made up and down the tree branches not across the branches.
3. All CALLS must be via the BLWP instruction.
4. No code is written in the program for overlaying. The overlay structure is specified in the LINKER control file and the LINKER inserts the code necessary to perform the overlay mapping.
5. The CALL stub for each overlayed subroutine is 16 bytes. CALLs within an overlay or towards the root of the tree have no overhead.
6. The overlay manager code inserted into the root segment is 60 bytes in size.

7. The mapping hardware works on 4K blocks in the >2000 to >3FFF and >A000 to >FFFF addressing ranges giving a maximum of a root section and 7 levels of overlay. Minimum size of an overlay is 4K.
8. The overlays must occur at 4K boundaries. The LINKER automatically adjusts section sizes rounding them up to the next 4K boundary.
9. The lengths of the various sections at a single overlay level do not have to be the same. The total depth of the CALL tree is 24K plus 8K.
10. Each overlay section consists of one or more object code files plus possible routines from libraries. That is, an overlay section can contain more than one subroutine.
11. The same subroutine code can be placed in different overlays on different branches. REFs are resolved up and down branches of the tree not across the call tree.
12. The LIBRARY statement must be used in each overlay section if resolution of REFs from the library is desired.

SUBROUTINE LIBRARIES

On the LINKER distribution disk is a library of object routines, RAGLIB. This is the same library distributed with the TI 99/4A versions of LINKER. It is for use with "standard" E/A Option 5 programs.

With the AEMS card is shipped a Library disk with a linker library, AEMSLIB, which contains routines specifically written for the AEMS.

RAGLIB Routines

The RAGLIB library contains the following routines:

```
DSRLNK Device Service Routine Link
GPLLNK GPL Subroutine Link
KSCAN  Keyboard Scan
XMLLNK Link to ROM routines
VMBR   VDP Multi Byte Read
VMBW   VDP Multi Byte Write
VSBR   VDP Single Byte Read
VSBW   VDP Single Byte Write
VWTR   VDP Write To Register
```

Which perform the same functions as the routines described in the Editor/ Assembler or Mini Memory manuals. The routines are all separate so that when the library is searched by LINKER only those routines that are actually used will be loaded. These routines (in particular GPLLNK) will work no matter which cartridge is used to load the memory image program. They are all relocatable object modules, and are not necessarily loaded into low memory as they are by E/A.

Note also, that the information about the DSR routine left in memory by the E/A and MM DSRLNK routines is not left by the library DSRLNK.

SYSTEM INFORMATION

The following subsections give details of some system information as handled by the LINKER. This information is not necessary for use of the LINKER, but may be interesting to Assembler programmers.

Object Files

Only the following tags in a tagged object can be processed:

>01	compressed object begin
0	uncompressed object begin
1	absolute program entry point
2	relocatable program entry point
3	REF in relocatable code
4	REF in absolute code
5	DEF relocatable
6	DEF absolute
7	checksum
8	checksum ignore
9	load address absolute
A	load address relocatable
B	absolute data
C	relocatable data
F	end of record
G	complex relocatable value
H	not currently implemented
I	COMMON definition and size

Note that the checksum, tag 7, is always ignored. The TI Assembler can produce other tags due to DORG, PSEG, DSEG and CSEG statements, but since these are not documented they cannot be processed.

Program Files

There are two types of program files, the standard memory image program file as defined by TI and extended by Miller's Graphics for GRAM program files; and the AEMS program file as defined in this document. The LINKER can produce either type of file. The AEMS format is required if a program makes use of overlays. The first word of either type of program file is a "flag word" that indicates what is contained in the file. The flag word is defined as follows.

Byte 1 - >FF This is not the last segment of the program.
>00 This is the last segment of the program.

Byte 2 - >FF Segment is to be loaded into CPU RAM.
>00 Segment is to be loaded into CPU RAM. Note that >00 and >FF are the same for compatibility to the TI standard.
>01 Load into GRAM 0 at >0000.
>02 Load into GRAM 1 at >2000.
>03 Load into GRAM 2 at >4000.
>04 Load into GRAM 3 at >6000.
>05 Load into GRAM 4 at >8000.
>06 Load into GRAM 5 at >A000.
>07 Load into GRAM 6 at >C000.
>08 Load into GRAM 7 at >E000.
>09 Load into ROM BANK 1 at >6000.
>0A Load into ROM BANK 2 at >6000.
>A0 AEMS non-overlay program or overlay root.
>A1 AEMS overlay segment.

Standard program files consist of the exact contents of memory written to a disk. A memory image program may be split into two or more segments depending upon the length of the program. Each of the segments is written to a separate disk file. Option 3 of TI WRITER will load a segment with a maximum size of >2000 while Option 5 of Editor/Assembler will load a segment with maximum size of >2400 bytes. LINKER produces segments with a maximum length of >2000. The name of the first segment is specified and the names for all following segments is derived by adding 1 to the last byte of the name of the previous segment. Each memory image file has a 3 word (or 6 bytes) header that indicates to a loader where in memory to load the program. The header is:

WORD 1 Flag word as described above.
WORD 2 Length of code in this segment in bytes. Note that the length of the segment on disk is 6 bytes more to include the 6 bytes of header information.
WORD 3 Address at which this segment is to be loaded.

NOTE that the length of the memory image program may not be the same as the length of the tagged object program. This can be caused by a work area that has no code or data loaded into it occurring at the end of the program. LINKER will not waste disk space writing out an area that you have not filled with code or data. In fact, if your program contains areas inside the program that are unused and which exceed 2048 bytes in size LINKER will break the program at that point and create two separate segments.

The AEMS Program File format is a bit more complicated than the standard in order to accommodate the much larger memory size and features of the AEMS system. An AEMS program is "page relocatable". That is, the parts of the program can be loaded into any contiguous block of pages in extended RAM. This page relocation allows the possibility of more than one program being loaded into memory at the same time.

The header for an AEMS program or root section of an overlay program is as follows.

WORD 1 Flag >FFA0 or >00A0.
WORD 2 Length of code in this segment in bytes.
WORD 3 Address at which this segment is to be loaded.
WORD 4 Total number of pages of memory required for the overlay program. This number includes a full 32K or 8 pages for the root section of an overlay program or is exactly 8 for a non-overlay program.
WORD 5 Entry address of the program. This word is repeated in every segment of a program.
WORD 6 Number of bytes in the "page relocation table" which begins at WORD 7. This may be zero.
WORD 7 If there are page relocation entries (WORD 5 is non-zero) then each word in the table is the address of a word containing a "relative page number" which is to be relocated.

Note that there may be more than one segment to the root section in a program and that the root section may be loaded at any address in the 4A's 64K address space. If there is more than one root segment WORD 4 (number of pages required) will be identical in all segments. It is redundant in all but the first segment.

The header for an AEMS overlay segment is as follows.

WORD 1 Flag >FFA1 or >00A1.
WORD 2 Length of code in this segment in bytes.
WORD 3 Address of this segment when it is mapped in for execution. The last 12 bits of this address is the offset within the relative page at which this segment is loaded.
WORD 4 Relative page number of this segment.
WORD 5 Entry address of the program.
WORD 6 Number of bytes in the "page relocation table" which begins at WORD 7. This may be zero.
WORD 7 If there are page relocation entries (WORD 5 is non-zero) then each word in the table is the address of a word containing a "relative page number" which is to be relocated.

Note that an overlay section may be more than one page in length and that there may be more than one file segment for an overlay section.

Overlay Code

The "Overlay Manager" code shown below is added to the root section of a program by the LINKER. Note that this code does not allow a single overlay section to be split between the low and high memory blocks.

```
* Map in N sequential pages and
* simulate BLWP call to subroutine
*
OVMGR  SBO   0           enable mapper regs
       MOV   *R11+,R10   Get N, # pages
       MOV   *R11+,R9    Get 1st mapper reg
       MOV   *R11+,R7    Get 1st page #
OVMGR2 MOV   R7,*R9      Set mapper reg
       AI    R9,>0010    Incr to next register
       AI    R7,onepage  Incr page #
       DEC   R10         Loop for N pages
       JGT   OVMGR2
       SBZ   0           Disable mapper regs
       MOV   *R11,*R11   Get real BLWP vector
       MOV   *R11+,R7    Get WSP
       MOV   *R11,R9     Get branch addr
       MOV   R13,@26(R7) Simulate BLWP
       MOV   R14,@28(R7)
       MOV   R15,@30(R7)
OVMGRW EQU   $-12        OVMGR workspace
* CALL user subroutine
       LWPI  0           R6,R7 Load user WSP
       B     @0          R8,R9 Go to user sub
       BSS   2           R10
       BSS   2           R11
       DATA  >1E00       R12 Mapper CRU addr
       BSS   6           R13-R15
```

Shown below is the "stub" inserted by the LINKER for each subroutine call that causes an overlay. Note that the return is direct to the calling BLWP, and that the mapped in pages remain mapped in on return.

```
* Stub for calling overlayed subroutine
* Called via BLWB  @OSUB
OSUB   DATA  OVMGRW      Overlay Manager WSP
       DATA  $+2
       BL    @OVMGR      Call manager
       DATA  N           # pages in overlay
       DATA  >40xx       1st mapper reg addr
       DATA  n           1st page number
       DATA  sub         real BLWP vector
```

AEMS Mapper

AEMS uses the Texas Instruments SN74LS612 memory mapper chip along with additional logic to map the low and high memory addresses from a 16 bit address to a 23 bit address. A 23 bit address accommodates a 4 Megabyte memory.

This mapping is done by splitting the TI 99/4A 16 bit address into two parts: a 4 bit page number and a 12 bit page offset. The 12 bit page offset gives a 4K page. The 4 bit page number is used to select a "mapper register" containing an 11 bit extended page

number. The 11 bit extended page number is combined with the original 12 bit page offset to give a 23 bit expanded memory address.

The mapper can be inactive or active. When the mapper is inactive the TI 99/4A will operate as though it had an ordinary 32K memory card. At power on, the mapper is inactive.

The mapper is activated by setting a CRU bit (>1E02) to one. When active, only addresses >2000 to >3FFF and >A000 to >FFFF are mapped. Mapping can be turned off by setting the CRU bit to zero.

The mapper has 8 active "registers" that specify which pages are mapped into the TI 99/4A's address space. In order to access these registers (for write or read) the access must be enabled by setting CRU bit >1E00 to one. Access is disabled by setting the CRU bit to zero. The mapper registers are accessed by normal 9900 instructions at the following addresses:

Register Number	Access at Address	Maps Page in at
2	>4004	>2000
3	>4006	>3000
A	>4014	>A000
B	>4016	>B000
C	>4018	>C000
D	>401A	>D000
E	>401C	>E000
F	>401E	>F000

The access addresses will respond to instructions just like a normal word of RAM. Typical code for operation of the mapper is shown below.

```
* Load Mapper Registers
       LI    R12,>1E00     CRU base address
       SBO   0             Enable register access
       LI    R0,20         Page # 20
       MOV   R0,@>4014     Map page in at >A000
       LI    R0,50         Page # 50
       MOV   R0,@>4016     Map page in at >B000
       SBZ   0             Disable register access
*
       SBO   1             Turn mapper on
       MOV   @>A000,R0     Get 1st word page 20
       MOV   @>B002,R1     Get 2nd word page 50
       SBZ   1             Turn mapper off
```

Only in special cases should the programmer have to manipulate the mapper or its registers directly. Assembler macros and library routines are provided for most routine tasks.

THE AEMS LOADER

The AEMS loader, permanently loaded into RAM by the AEMS boot program, is the heart of the AEMS system. Its main function is to load and execute programs. It also contains system routines for management of the extended memory pages and some system routines that make programming for the AEMS system easier.

The AEMS loader does not occupy space in the 4A's address space. The loader, its data areas and its system routines are mapped in and out of the address space as required.

The AEMS system controls the use of RAM. The following table shows the use of the extended RAM pages.

Page #	Use
>00	System. AEMSLOAD - Mapped in at >A000
>01	System. AEMSLOAD - Mapped in at >B000
>02	Reserved. Low RAM >2000 when mapper inactive
>03	Reserved. Low RAM >3000 when mapper inactive
>04	System
>05	System
>06	System
>07	System
>08	System
>09	System
>0A	Reserved. High RAM >A000 when mapper inactive
>0B	Reserved. High RAM >B000 when mapper inactive
>0C	Reserved. High RAM >C000 when mapper inactive
>0D	Reserved. High RAM >D000 when mapper inactive
>0E	Reserved. High RAM >E000 when mapper inactive
>0F	Reserved. High RAM >F000 when mapper inactive
>10	RAM mapped as required
>11	RAM mapped as required
etc	...
etc	...

EXAMPLES

The following examples show some typical uses of the LINKER. In each case, a situation is set up then the LINKER control file and the LINKER prompts are shown. In the design and coding of LINKER every effort was made to keep everything as general as possible. Because of this the LINKER can do a lot of useful things (and probably a lot of things that are not useful). The examples show some "normal" uses. With a little thought and practice you can soon develop some tricks that seem useful to you.

While experimenting with LINKER you should select all the options so that a listing is produced showing exactly what the LINKER did with your program.

EXAMPLE 1

This is the simplest case. Suppose you have a single assembler language program that is complete and requires no subroutines. The program is assembled into a relocatable tagged object module into file "DSK1.EXAMPLE1/O". You want to make a memory image program that loads into high memory (i.e. at >A000). The program is to be placed in the file "DSK1.PROGRAM1".

LINKER Control File

The object file is the control file in this case.

LINKER Input Fields

```
Printer:PIO
Library:
Options:SML
Control:DSK1.EXAMPLE1/O
Program:DSK1.PROGRAM1

Link More?
```

The requested program file is created and written to file DSK1.PROGRAM1 in the Standard E/A Option 5 format. A memory map of the program is printed along with a message giving the header information of the memory image file.

EXAMPLE 2

Suppose you have a single assembler language program that has REFs to the routines VMBR and VMBW (from the library supplied on the LINKER disk). The program is assembled as a relocatable tagged object module into file "DSK1.EXAMPLE2/O". You want to make a memory image program that loads into the high memory expansion (i.e. at >A000). The program is to be placed in the file "DSK1.PROGRAM2". You want a listing with all the optional data LINKER can provide.

LINKER Control File

The object file is the control file in this case.

LINKER Input Fields

```
     Printer:PIO
     Library:DSK1.RAGLIB
     Options:MLFX
     Control:DSK1.EXAMPLE2/O
     Program:DSK1.PROGRAM2

     Link More?
```

The requested program file is created and written to file DSK1.PROGRAM2. The printed listing contains full information about the object modules processed and the memory image program produced.

EXAMPLE 3

Suppose your program consists of a main program and two subroutines all three of which have been separately assembled into files DSK2.MAIN3/O, DSK2.SUBA3/O and DSK2.SUBB3/O. This program also has REFs to the routines VMBR and VMBW (from the library supplied on the LINKER disk). You want to make a program that loads into high memory (i.e. at >A000). The program is to be placed in the file "DSK2.PROGRAM3". You want a listing with all the optional data LINKER can provide.

LINKER Control File (EXAMPLE3/L)

```
*
* CREATE PROGRAM 3
*
LOAD DSK2.MAIN3/O
LOAD DSK2.SUBA3/O
LOAD DSK2.SUBB3/O
```

LINKER Input Fields

```
Printer:PIO
Library:DSK1.RAGLIB
Options:MLFX
Control:DSK2.EXAMPLE3/L
Program:DSK2.PROGRAM3

Link More?
```

The requested program file is created and written to file DSK1.PROGRAM3. A listing is printed showing everything about the linked program.

EXAMPLE 4

Suppose you have a single assembler language program that has REFs to the routines VMBR and VMBW (from the library supplied on the LINKER disk). The program is assembled into a relocatable tagged object module into file "DSK1.EXAMPLE4/O". You want to make a program file that loads into low memory (i.e. at >2000). The program is to be placed in the file "DSK1.PROGRAM4". You want a listing with all the optional data LINKER can provide. Suppose also that you have a single disk system.

In this case you need linker control statements because you want your program loaded in low memory. Suppose you have created the control file with an editor on the same disk as the object file named "DSK1.EXAMPLE4/L".

LINKER Control File

```
* INTERCHANGE BLOCK 1 AND 2
* TO MAKE PROGRAM LOAD IN LOW MEMORY FIRST
BLOCK 1,>2000,>2000    LOW MEMORY
BLOCK 2,>A000,>6000    HIGH MEMORY
LOAD DSK*.EXAMPLE4/O
```

LINKER Input Fields

```
Printer:PIO.
Library:*.RAGLIB
Options:MLFX
Control:DSK1.EXAMPLE4/L
Program:*.PROGRAM4

Link More?
```

The requested program file is created and written to file DSK1.PROGRAM4.

Note the "shorthand" method of entering the "device" part of the Library and Program file names, indicating they are on the same device as the control file.

EXAMPLE 5

Suppose you have an assembler program in object form in file DSK2.EXAMPLE5/O. This program begins execution at START which is defined by a DEF in the program, but which is not the first byte of the program. You want to make a program that loads into high memory (i.e. at >A000). The memory image program is to be placed in the file "DSK1.PROGRAM5". You want a listing with all the optional data LINKER can provide.

LINKER Control File (EXAMPLE5/L)

```
*
*    CREATE PROGRAM 5
*
LOAD DSK1.EXAMPLE5/O.
ENTRY START           TELL LINKER WHERE THE ENTRY IS
```

LINKER Input Fields

```
Printer:PIO
Library:
Options:SMLFX
Control:DSK2.EXAMPLE5/L
Program:DSK1.PROGRAM5

Link More?
```

Because the program begins execution at other than the first byte and because the "S" option was specified to force a standard E/A Option 5 program file, LINKER creates two segments for the program. The first segment begins at START and ends at the end of the program; it is written to file DSK1.PROGRAM5. The second segment begins at the first byte of the program and ends just before START; it is written to file DSK1.PROGRAM6.

EXAMPLE 6

Suppose, the same situation as example 5, that is, you have an assembler program in object form in file DSK2.EXAMPLE6/O. This program begins execution at START which is defined by a DEF in the program, but which is not the first byte of the program. You want to make a program that loads into high memory (i.e. at >A000). In this case you want to make the program file be a single segment (or file). The program is to be placed in the file "DSK1.PROGRAM6". You want a listing with all the optional data LINKER can provide.

LINKER Control File (EXAMPLE6/L)

```
*    CREATE PROGRAM 6
*
*  IN ORDER TO CREATE A SINGLE SEGMENT,
*  WE PATCH A BRANCH AT >A000 TO THE REAL
*  ENTRY POINT OF THE PROGRAM, "START".
*
BLOCK 1,>A004,>5FFC   LEAVE 4 BYTES FOR
*                     THE PATCH.
LOAD DSK2.EXAMPLE6/O
PATCH >A000,>0460     BRANCH INSTRUCTION
PATCH >A002,START     ADDRESS FOR BRANCH.
ENTRY >A000           TELL LINKER ABOUT THE ENTRY
```

LINKER Input Fields

```
Printer:PIO
Library:
Options:SMLFX
Control:DSK1.EXAMPLE6/L
Program:DSK1.PROGRAM6

Link More?
```

Note that the PATCH statements must come after the object module is loaded so that we can refer to the symbol "START".

EXAMPLE 7

It is sometimes convenient when programming a large program to divide it into smaller pieces. Often when this is done you need all the separate pieces to refer to the same work/data area. This can be done by making the work/data area absolute code so that all pieces know the address of the data. A problem can arise when you have a mixture of absolute and relocatable code like this. This problem occurs because the loaders for the TI 99/4A do not keep track of absolute code and may load a relocatable module over top of your common work/data area.

Making use of the LINKER's BLOCK structure you can overcome this problem. At the same time, LINKER keeps track of all code or data loaded and will automatically produce whatever segments are required for the program.

Suppose you have this situation. You have a main program (MAIN7/O), two subroutines (SUBA7/O and SUBB7/O) and an absolute work/data area (WORK7/O). The work/data area was assembled at absolute address >F000 to >FFFF.

LINKER Control File (EXAMPLE7/L)

```
*    CREATE PROGRAM 7
*
*  CHANGE THE LINKER MEMORY BLOCKS TO
*  PREVENT OVERLAY OF ABSOLUTE WORK AREA
*
BLOCK 1,>A000,>5000  LEAVE OUT >F000 UP
LOAD DSK2.MAIN7/O
LOAD DSK2.SUBA7/O
LOAD DSK2.SUBB7/O
```

LINKER Input Fields

```
Printer:DSK1.LISTING
Library:
Options:MLFX
Control:DSK1.EXAMPLE7/L
Program:DSK1.PROGRAM7

Link More?
```

Note in this case we sent the listing to disk in file DSK1.LISTING. Note also that the COMMON facilities of the AEMS Macro Assembler and the AEMS Linker make the use of this method of using "common data" areas obsolete. The preferred and easier way now is to define a COMMON in each assembly. This common area then will be relocatable and the LINKER will allocate memory for it wherever it can.

EXAMPLE 8

Suppose you have an object program in file DSK1.EXAMPLE8/O. You want to make a memory image program that loads into the RAM in the cartridge slot (i.e. at >6000). The program is to be placed in the file DSK1.PROGRAM8". No listing is required.

LINKER Control File (EXAMPLE8/L)

```
*
*     CREATE PROGRAM8
*
*  MAKE PROGRAM LOAD IN CARTRIDGE RAM
*
BLOCK 1,>6000,>2000   CARTRIDGE SLOT RAM
LOAD DSK*.EXAMPLE8/O
```

LINKER Input Fields

```
Printer:PIO
Library:
Options:S
Control:DSK1.EXAMPLE8/L
Program:*.PROGRAM8

Link More?
```

The requested program file is created and written to file DSK1.PROGRAM8. The program will load in the RAM at address >6000 to >7FFF.

EXAMPLE 9

This example shows how to build an overlay program. The program is the example used in the section "OVERLAY PROGRAMS". The nine subprograms are assembled into object files: MAIN/O, INIT/O, PASS1/O, PASS2/O, ENDUP/O, GETREC/O, BLDTBL/O, GETTBL/O and PUTREC/O.

LINKER Control File (EXAMPLE9/L)

```
*
*     CREATE OVERLAY PROGRAM
*
ORIGIN >2000        Root segment in low memory
LOAD DSK*.MAIN/O    Root code
OVERLAY 1           1st Level of Overlay
LOAD DSK*.INIT/O
OVERLAY 1           At 1st level again
LOAD DSK*.PASS1/O
OVERLAY 2           2nd level of PASS1
LOAD DSK*.GETREC/O
OVERLAY 2           At 2nd level PASS1 again
LOAD DSK*.BLDTBL/O
OVERLAY 1           At 1st level again
LOAD DSK*.PASS2
OVERLAY 2           2nd level of PASS2
LOAD DSK*.GETTBL/O
OVERLAY 2           2nd level PASS2 again
LOAD DSK*.PUTREC/O
OVERLAY 1           At 1st level again
LOAD ENDUP
```

LINKER Input Fields

```
Printer:PIO
Library:
Options:OFLM
Control:DSK1.EXAMPLE9/L
Program:DSK1.PROGRAM9A

Link More?
```

The requested overlay program is created and written to files DSK1.PROGRAM9A, DSK1.PROGRAM9B, DSK1.PROGRAM9C, DSK1.PROGRAM9D, DSK1.PROGRAM9E, DSK1.PROGRAM9F and DSK1.PROGRAM9G. Note that the overlay option "O" was used.

EXAMPLE 10

This example is the same as Example 9 except that we will assume that most of the subprograms use some library calls. Assume they all call VMBW and VMBR and that GETREC and PUTREC call DSRLNK. The nine subprograms are assembled into object files: MAIN/O, INIT/O, PASS1/O, PASS2/O, ENDUP/O, GETREC/O, BLDTBL/O, GETTBL/O and PUTREC/O.

LINKER Control File (EXAMPL10/L)

```
*     CREATE OVERLAY PROGRAM
ORIGIN >2000        Root segment in low memory
LOAD DSK*.MAIN/O    Root code
* Load common routines in root segment
LOAD DSK1.AEMSLIB,VMBR
LOAD DSK1.AEMSLIB,VMBW
OVERLAY 1           1st Level of Overlay
LOAD DSK*.INIT/O
OVERLAY 1           At 1st level again
LOAD DSK*.PASS1/O
OVERLAY 2           2nd level of PASS1
LOAD DSK*.GETREC/O
LOAD DSK1.AEMSLIB,DSRLNK  Load library routine
OVERLAY 2           At 2nd level PASS1 again
LOAD DSK*.BLDTBL/O
OVERLAY 1           At 1st level again
LOAD DSK*.PASS2
OVERLAY 2           2nd level of PASS2
LOAD DSK*.GETTBL/O
OVERLAY 2           2nd level PASS2 again
LOAD DSK*.PUTREC/O.
LOAD DSK1.AEMSLIB,DSRLNK  Load library routine
OVERLAY 1           At 1st level again
LOAD ENDUP
```

LINKER Input Fields

```
Printer:PIO
Library:
Options:OFLM
Control:DSK1.EXAMPL10/L
Program:*.PROGRM10A

Link More?
```

The requested overlay program is created and written to files DSK1.PROGRM10A, DSK1.PROGRM10B, DSK1.PROGRM10C, DSK1.PROGRM10D, DSK1.PROGRM10E, DSK1.PROGRM10F and DSK1.PROGRM10G. Note that the overlay option "O" was used. Note also that DSRLNK was loaded into two different branches of the overlay.