GenLINK
v1.00

Reference guide.

NOTE: GenLink seems to be more efficient then QDL V2.0.
GenLink made a single large file and it was in total
a little smaller then the total of the two files
created by QDL. (QDL must have a control file
as input).

CONTENTS

INTRODUCTION

GenLINK is useful for any person who wants to create programs in image format for MDOS or the TI-99/4a. It is also useful for people who wish to create GPL programs in image format.

GenLINK is a program which translates "Tagged Object" files into "Program Image" files.

GenLINK's primary features are:

1) runs from MDOS
2) no restrictions on size of image programs
3) can be run both as a batch file and interactively
4) can be used to create "SYSTEM/SYS" style program images.
5) will create symbol files for a symbolic debugger
6) complex equations using REF/DEF symbols can be used instead of hard-coded constants.
7) can resolve REF symbols from tagged object library files.
8) can build program images which aren't contiguous in memory.
9) includes libraries for MDOS and TI-99/4a programmers.
10) can be used to create program overlays which load at common address, will resolve REF/DEF between modules at the same address.
11) REF/DEF symbols can have be up to 31 characters in length.

OVERVIEW

GenLINK uses five different types of files during the course of its execution, these will be described briefly now.

"LINK" is the MDOS program file which reads the control file (or keyboard input), tagged object files, tagged object libraries and creates the program image output files.

Control file (or keyboard input), this file contains commands which must be executed by GenLINK in order to create the program images files from the tagged object files and the tagged object libraries. A control file (if keyboard input is not used) must be created by the user using a standard text editor which saves files in Dis/Var 80 format.

Tagged Object file(s), this type of file is usually created by an assembler (or compiler) and contains all of the information needed to build a part of the program image

file. It may include information to be passed on to a symbolic debugger.

Tagged Object library(s), this type of file is a collection of tagged object files which have been placed into one larger file via a Librarian program (such as GenLIB.) It generally contains many simple, often-used, subroutines. This file also has an index which enables GenLINK to quickly load only the sections which are needed to include referenced sub-programs. GenLINK comes with two predefined libraries, "LIB_MDOS" with routines useful to MDOS programmers, and "LIB_4A" with routines useful to TI-99/4a programmers.

Debugger symbol file, this type of file contains information needed by a symbolic debugger. It has records for different types of symbols with line numbers and filenames for each symbol.

USING GenLINK

GenLINK is executed from an MDOS command line or from within an MDOS batch file.

You must perform the following actions before using GenLINK:

First, MDOS must be able to find the file "LINK" somewhere in your current command path (set with the "PATH" command in MDOS.)

Second, you must create all tagged object files which are to be included in the resultant program image file.

Third, GenLINK must be able to write to the program image and (optional) debugger symbol files. This means that your destination disk can not be write protected, and that it must have enough free sectors to allow the files to be written.

Recommended (optional), you should create a Dis/Var 80 control file for the linker.

Continuing with the assumption that the previous conditions have been met, GenLINK is invoked from MDOS with the following command format:

LINK [control_file] (brackets indicate optional items)

If no control file was specified, GenLINK will prompt you for keyboard input. Otherwise, it will read one line at a time from the control file and process each line as if it had been typed from the keyboard.

CREATING A CONTROL FILE

Control files may be created with any editor which can save files in a Dis/Var 80 format. You may have one command per line in the file, and you may have comment lines. Each line in the control file must begin with a non-blank character for GenLINK to recognize it as a valid command.

Control files are most useful when you want to run the linker in a completely automatic mode, from a batch file, or as part of an update batch file created by GenMAKE.

"EXIT" must be the last command executed in the control file for completely automatic operation. If GenLINK does not find an "EXIT" command before it finds the end of the control file, it will prompt you for keyboard input.

Commonly used commands:

ADD

syntax: ADD object_filename

This is the command you will probably use the most.

The first function performed by the ADD command is to read the RORG length header from the object file. ADD then checks all defined memory blocks, in the order that the memory blocks were defined, and uses the first memory block it finds which has enough room for the RORG part of the object file. (You really don't need to worry about memory blocks if you are building simple MDOS programs, since GenLINK defines a default memory block which is suitable for most of your needs. Memory blocks are explain in the section on "fragmented images".)

After finding a memory block with enough room, the ADD command loads the entire object file into memory, and builds tables for all REF, DEF, FILE, and DEBUG symbols defined in the file. No REF symbols from other object modules are resolved yet. If there is an error during the execution of the ADD command, the tables of all symbols which had already been found in the file are flushed.

Once the ADD command successfully loads the file, it copies all of the newly defined symbols into tables which hold globally defined symbols during the link process. ADD then goes through the table of new DEF symbols, and checks to see if any of them would resolve outstanding REF symbols from any previously loaded module.

You will generally want to load the object file with a program's entry point first. The object file with the entry point must be small enough to fit in the first defined memory block, and the first word(s) of data in the object file must be executable instructions for the program's entry point (This is not applicable for GRAM and ROM image files.)

DEBUG

syntax: DEBUG debug_filename

The DEBUG command causes GenLINK to make a file containing all of the DEBUG and FILE symbols which were defined in previously loaded object modules.

LIBREF

syntax: LIBREF library_filename

The LIBREF command causes GenLINK to search for the name of each unresolved REF symbol (from its global REF table) in the index of the specified library file. If the symbol is found in the index of the library file, the portion of the library file which contains the symbol will be loaded in the manner described in the ADD command.

The LIBREF will search the index of the specified library until it recognizes that no symbol in the REF table is in the index. GenLINK will attempt to resolve REFs generated in loading part of the library by continuing to search the library index.

SAVEALL

syntax: SAVEALL image_filename[,type]

The SAVEALL command directs GenLINK to save all currently defined memory blocks, starting with the first memory block, to a sequence of files whose names begin with the specified image filename.

Each memory block will be saved to a separate image filename. If the amount of space used in any memory block is larger than the maximum file length for the image type specified, the block will be broken into smaller blocks which are all at least as small as the maximum allowed file length for that image type. MDOS fast-image format is the default for the SAVEALL command. GRAM image files are also fragmented at points where they would cross an 8k-byte boundary address.

The sequence of filenames generated by the SAVEALL command as it executes begins with the provided image filename. If there is more than one file in the sequence, each filename after the first file is generated by adding one to the ASCII value of the last character in the previous filename in the sequence. This sequence is not compatible with the sequence used by GRAM image files. (The proper sequence for GRAM image files may be generated by specifying the first filename to be "<filename>0", then renaming "<filename>0" to "<filename>" once the SAVEALL command is complete. This rename process may be done automatically in the control file by using the MDOS command of GenLINK.)

The "types" recognized by SAVEALL are:

0,1 MDOS fast-ram image.
max length = >3DFA, flag byte = 'F'

2,3 MDOS slow-ram image.
max length = >3DFA, flag byte = 'G'

4,5 TI-99/4A program image.
max length = >1FFA, flag byte = >00 | >FF

6,7 GRAM program image.
max length = >2000, flag byte = >01..>08

8,9 TI-99/4A rom bank 0 at >6000
max length = >2000, flag byte = >09

10,
11 TI-99/4A rom bank 1 at >6000
max length = >2000, flag byte = >0A

12,
13 TI-99/4A 4k rom bank at >6000 (like Mini-Memory cart)
max length = >1000, flag byte = >0B

General Commands:

COMMENTS

syntax: *<any text you want until end of line>

Any line whose first character is an asterisk, "*", is ignored by GenLINK. COMMENTS are useful for documenting various part of a control file, so that you can remember later on why you did things in a certain way. COMMENTS are also useful for explaining the logic of what you did to someone you will never meet, if you plan to distribute your control files to other people.

EXIT

syntax: EXIT

The EXIT command causes GenLINK to return to MDOS.

HELP

syntax: HELP

The HELP command provides a list of all commands recognized by GenLINK, and a brief summary of the correct syntax for the command.

MDOS

syntax: MDOS <mdos command string>

The MDOS command allows you to execute any internal command of the Command Line Interpreter. Functions such as DIR and DEL are most commonly used in this manner.

Commands for building fragmented images:

BLOCK

syntax: BLOCK <start address>,<end address>

The BLOCK command is used to define disjoint areas of memory where object files may be loaded. It is most useful for defining memory overlays, or images programs which make use of all available memory on the TI-99/4A.

The most common use of this (TI-99/4A programmers) is the following two lines:

```
BLOCK >A000,>FFD8
BLOCK >2000,>4000
```

These two lines set up an object code environment which allows you to use all of the 32k expansion memory for your image code.

Both the start_address and the end_address specified in the BLOCK command are 32-bit linker expressions. The BLOCK command only uses the least significant 16-bits of any linker expression as the boundaries for the block.

You are allowed to specify a block which overlaps other blocks you have defined. Users of GenLINK are invited to figure out some use for this capability.

Use of the BLOCK command in conjunction with the PAGE command allows you to easily define overlay code with REF/DEF symbols to other overlays at the same address (see examples.)

You must define a BLOCK for each range of addresses you plan to load with AORG code, so that the SAVEALL command knows to create an image of the memory containing the AORG code.

Blocks are prioritized for loading in the order you define the blocks.

You may view your current block definitions with the STATUS command.

CLEAR

syntax: CLEAR

The CLEAR command is most useful when you are building program overlays which reside at the same range of memory addresses.

For overlays, you'll need to use the PAGES command. The examples for overlays are included with the PAGES command.

Also, at times (not very often), you will have an application in which you want to save the blocks in a different order than that in which you defined them, or you may want to save areas of memory which weren't loaded because of your initial block definitions.

For these applications, you will want to redefine the block definitions between the load and save steps. The CLEAR command lets you accomplish this by erasing all of your current block definitions, so that you can define new ones.

The sequence of commands you might want to use is:

```
BLOCK a,b
BLOCK c,d
ADD   file
ADD   file
CLEAR
BLOCK e,f
BLOCK g,h
SAVEALL
```

The ADD command will only load code into the area defined by the ranges [a-b] and [c-d]. The SAVEALL command will save memory in the range [e-f] as image file(s), then will save memory in the range [g-h] as image file(s). The program entry point should be at address "e".

This command is most useful when you don't wish to place your program's entry point at the beginning of the program. You can do the following in such a situation:

```
BLOCK a,b
ADD   file
CLEAR
BLOCK ENTRY,b
BLOCK a,ENTRY
```

The previous procedure will cause the ENTRY address of the program to be the first address in the first image saved for the program.

COPY

syntax: COPY <source>,<dest>,<length>

In the COPY command, each of the specified values are 32-bit linker expressions. Only the least significant 16 bits of the "length" value are used as a byte count.

The source address specified is the address of the first byte of data to copy. The destination address is the address where the first byte copied is to be placed. The specified length is the number of bytes to be copied.

The COPY command correctly copies overlapping source and destination regions (which unfortunately means that it can't be used to "fill" memory with a byte pattern.)

You will probably not need to use source or destination addresses larger than 16-bits unless you understand how to use more than 64k while you are building overlays, or you want to build "SYSTEM/SYS" style program images.

PAGES

syntax: PAGES p0[,p1..p7]

The PAGES command is useful for building overlays and programs which are larger than 64k. Each page contains 8k bytes, and you can not specify more than eight pages (64k bytes) for loading with one PAGES command. Each page is specified with a linker expression, and the value of the linker expression can not exceed the number of memory pages free in MDOS at the time the linker is run.

Pages assigned in the list following the PAGES command are located at the following addresses:

1st page in list:	>0000
2nd page in list:	>2000
3rd page in list:	>4000
4th page in list:	>6000
5th page in list:	>8000
6th page in list:	>A000
7th page in list:	>C000
8th page in list:	>E000

You may declare any of the 8 pages to be "undefined" if you leave its place in the list empty. GenLINK will detect any attempt to load object code into an "undefined" page and report an error to you.

EXAMPLE: suppose you want to write a program which consists of one 8k header page, at address >0000 (which should have the common subroutines and code to call subroutines in various overlays), and three 16k overlay sections which get mapped in at >2000 before they get used. The size of the entire program is 7 pages (56k). You could link the program as follows:

```
*
* load header section
*
PAGES 0,
BLOCK >0400,>2000
ADD   HEADER-OBJ
*
* load the first overlay (at >2000)
*
PAGES ,1,2
BLOCK >2000,>6000
ADD   OVERLAY1-OBJ
*
* load the second overlay (at >2000)
*
PAGES ,3,4
BLOCK >2000,>6000
ADD   OVERLAY2-OBJ
*
* load the third overlay (at >2000)
*
PAGES ,5,6
BLOCK >2000,>6000
ADD   OVERLAY3-OBJ
*
* now to save it all as one linked chain
* map it in with a PAGES command, then
* declare a BLOCK which includes all of the data
* which needs to be saved
*
PAGES 0,1,2,3,4,5,6,
BLOCK >0400,>E000
SAVEALL
*
*
*
```

In the preceeding example, all REF/DEF symbols are resolved, regardless of which section they were defined in and which section(s) they were used in.

PSAVE

syntax: PSAVE image_filename,[type],[start],[stop]

PSAVE is a variation of the SAVEALL command which only saves one program segment at a time. You, as a programmer, are in control of how large each segment is and in what order they are to be loaded from disk. PSAVE also allows you to save SYSTEM/SYS style program images.

Each of the three optional parameters is a 32-bit linker expression. The length of an image saved with the PSAVE command, Stop minus Start, can not exceed the maximum length allowed for the Type specified.

The "types" recognized by PSAVE are:

0 MDOS fast-ram image, not last in chain of images.
max length = >3DFA, flag byte = 'F'

1 MDOS fast-ram image, last in chain of images.
max length = >3DFA, flag byte = 'F'

2 MDOS slow-ram image, not last in chain of images.
max length = >3DFA, flag byte = 'G'

3 MDOS slow-ram image, last in chain of images.
max length = >3DFA, flag byte = 'G'

4 TI-99/4A program image, not last in chain of images.
max length = >1FFA, flag byte = >FF

5 TI-99/4A program image, last in chain of images.
max length = >1FFA, flag byte = >00

6 GRAM program image, not last in chain of images.
max length = >2000, flag byte = >01..>08

7 GRAM program image, last in chain of images.
max length = >2000, flag byte = >01..>08

8 TI-99/4A rom bank 0 at >6000, not last in chain of images.
max length = >2000, flag byte = >09

9 TI-99/4A rom bank 0 at >6000, last in chain of images.
max length = >2000, flag byte = >09

10 TI-99/4A rom bank 1 at >6000, not last in chain of images.
max length = >2000, flag byte = >0A

11 TI-99/4A rom bank 1 at >6000, last in chain of images.
max length = >2000, flag byte = >0A

12 TI-99/4A 4k rom bank at >6000 (like Mini-Memory cart), not last in chain of images.
max length = >1000, flag byte = >0B

13 TI-99/4A 4k rom bank at >6000 (like Mini-Memory cart), last in chain of images.
max length = >1000, flag byte = >0B

14 Program image of any length, no header added to data, like SYSTEM/SYS, but there are certainly other uses.
A 90k byte SYSTEM/SYS is saved with a command like:

PSAVE "SYSTEM/SYS",14,>00000,>16500

RESET

syntax: RESET

The RESET command causes GenLINK to re-initialize all of its internal tables and REF/DEF symbol information, as if GenLINK had just been freshly loaded by MDOS. This is useful if you wish to create several different programs with the same linker control file.

Commands which provide special information:

EVAL

syntax: EVAL linker-expression

The EVAL command simply prints out the value of the linker expression you specified. This is most useful for displaying the values of various symbols in the DEF table. It is also useful for calculating the size of various parts of the program (if you subtract the ending address from the beginning address of the segment.)

LIST

syntax: LIST list-filename

This command opens the specified list file, and all subsequent output which GenLINK displays on the screen is also echoed into the list file for later inspection by you.

STATUS

syntax: STATUS

The STATUS command lists information about usage of all memory blocks defined with the BLOCK command since the last CLEAR command was issued, as well as how much space is left in the block with the most remaining space. It also tells you how many symbols there are in the global DEF table.

For each currently defined block of memory, the STATUS command lists the following information:

1) Start address of block
2) First free address in block for more data.
3) Last address in block

SYMTAB

syntax: SYMTAB

The SYMTAB command causes GenLINK to display the value and name of each symbol in the global DEF table.

UNDEF

syntax: UNDEF

The UNDEF command causes GenLINK to display the address and name of each symbol in the global REF table. You would use this command to determine which REFs have not yet been resolved.

Linker expressions

Many commands in GenLINK which use numeric parameters will allow you to use a "linker expression" to specify the numeric value.

A linker expression always returns a 32-bit value and has the following form:

[value [operator value] ... [operator value]]

All linker expressions are evaluated from left to right. The expression: "1+2*5" has a value of 15, not 11.

If the linker expression is null, a value of zero will be returned.

The following are the valid "value" fields in a linker expression:

DEF symbol:	name from global DEF table	
Hex constant:	">hex digits",	range: >0000 to >ffffffff
Decimal constant:	"decimal digits",	range: 0 to 65535

The following operators can be used in a linker expression: "+" (addition), "-" subtraction, "*" (multiply two 16-bit values), "/" (divide a 32-bit value by a 16-bit value with a 16-bit result.)

Object file format:

The first character in an object file must be an identifier tag, which must be >01 to indicate compressed object code or "0" to indicate uncompressed object code. Each byte in uncompressed object code requires two hex digits to indicate its value, while each byte in compressed object code is used "as-is" with no extra interpretation.

GenLINK reconizes the following tags:

(items in curly brackets may be compressed or uncompressed values, all other items must be single-byte characters in the object file.)

>01 | "0" {2 byte length} <8 char identifier>

The 2 byte length indicates the number of RORG memory locations which must be reserved for the object file. It is also used to determine if there is enough room in defined memory blocks to allow the object file to be completely loaded.

These tags are invalid if they are found at any point in the object file after the first byte.

start_address = first_free_address,
in block with sufficient space

first_free_address = start_address +
round_to_word(length)

linker_data_pointer = start_address

"1" | "2" {2 byte auto-start address}

These tags are ignored by the linker since auto-start addresses are only useful for a dynamic object code loader.

"3" {2 byte relocatable REF chain header} <symbol>
"4" {2 byte absolute REF chain header} <symbol>

These tags are used to define symbols which are to be resolved from other object modules. Each REF symbol in a file must be placed into the object file as a linked-list of addresses, the data at each address in the object file must

contain a pointer to the next location which needs to be patched when the REF is resolved. The last location to be patched must have a value of >0000 in the object file.

The following algorithm is used to resolve REFerenced symbols:

```
location = header
while (location != 0)
  do {
        temp = *location;
        *location = DEF symbol value;
        location = temp
     }
```

The "symbol" used in the REF may have one of two forms, depending on the number of characters in the symbol name.

If the symbol name has six or fewer characters, it must be entered into the object file in a field six characters wide, padded on the right with spaces if it is shorter than six characters to start with.

If the symbol name has seven to thirty-one characters, it must be entered into the object file as <length><characters>, where <length> is a single character with a value of >07 to >1f.

"5" {2 byte relocatable address} <symbol>
"6" {2 byte absolute address} <symbol>

These tags are used to define symbols for use in other object modules. The 2 byte address specified is used as the value of the symbol.

The symbol's name must be constructed as described under tag "3" and "4".

"7" | "8" {2 byte checksum}

These two tags are ignored by GenLINK.

"9" {2 byte absolute program counter}
"A" {2 byte relocatable program counter}

These two tags are used to specify the address at which subsequent data in the object file will begin loading.

```
"9":  linker_data_pointer = value
"A":  linker_data_pointer = value + start_addr
```

"B" {2 byte absolute data word}
"C" {2 byte relocatable data word}

```
round_to_word(linker_data_pointer);
if (tag == "B")
   *linker_data_pointer = data
else
   *linker_data_pointer =
    data + linker_data_pointer
linker_data_pointer += 2;
```

"D" {single byte value}

```
*linker_data_pointer++ = byte;
```

"F"

This tag cause GenLINK to stop processing of tags in the current record of the object file, and skip to the first byte in the next record.

"G" {line #}{file #}{reloc address}{type}<name>
"H" {line #}{file #}{abs address}{type}<name>

The line number (2 bytes) following these tags should be the line number within the source code which corresponds to the address (2 bytes) specified with the tag. This information will be placed into the debugger symbol file with the DEBUG command.

The file number (2 bytes) following these tags should be the same as the file number specified with a file record in the same object file. The file number is altered before sending this information to the debugger symbol file, so that it does not conflict with files by the same number in other object modules.

The "type" field (2 bytes) is passed directly to the debugger symbol file. GenASM only produces type ">0000" symbols, which indicates that a simple data address corresponds to the name.

A compiler would want to produce line records which don't correspond to any particular data item, these could be specified with a type of ">0001" and a blank name field. (Interpretation of the "type" field is left to the debugger, a debugger for a compiler might want different types to be specified for various data, such as FLOAT, DOUBLE, STRING, char *, etc.)

The name field consists of one character with a value of >01 to >1f for a length, followed by the number of characters indicated in the length character.

"I" {file #}<file name>

The file number (2 bytes) will be used by a debugger to determine the filename of symbols with the same file number. The file number is altered before sending this information to the debugger symbol file, so that it does not conflict with files by the same number in other object modules.

The name field consists of one character with a value of >01 to >1f for a length, followed by the number of characters indicated in the length character.

":"

Loading of an object module is complete when GenLINK encounters this tag as the first character within an object record.

Library file format:

Each library file has three major parts:

1) object code sections
2) hashed symbol index
3) free space list

The first record (record 0) of a library file has the following format:

": " <14 byte id> hash[31] next_free[1] line_count[1]

The line count (2 bytes) indicates how many contiguous unused lines there are in the library file, starting with the next record (record 1).

The next_free pointer (2 bytes) is the record number of the next record in the free-space management list, this is used when the librarian is searching for a free block with enough space to add a new object file to the library. If the next_free pointer is zero, there are no more records in the free space list.

The hash array contains 31 pointers (2 bytes each) which are the record numbers of the first index record in a hashed list of symbols. The index into the hash array is calculated by taking the sum of all characters in a symbol MOD 31. Any pointer with a value of zero indicates a hash list with no symbols in it.

Records in the hash list have the following form:

": " next_hash_record { object_pointer symbol_name }

(The portion in squigly brackets can be repeated many times in the record.)

The "next_hash_record" is a pointer (2 byte record number) to the next record which contains symbols in the current hash list. If the pointer is zero, the current record is the end of the hash list.

The "object_pointer" is a 2 byte record number which specifies the first record of the object module which will resolve the following symbol name. If the object_pointer is zero, there are no more symbols to be examined in the current hash record.

The name field consists of one character with a value of >01 to >1f for a length, followed by the number of characters indicated in the length character. If the

length of the symbol is even, the symbol name will be followed by a single pad character so that the next symbol section starts on an even byte boundary within the hash record.

Records in the free space list have the following format:

": "<74 garbage characters> next_free[1] line_count[1]

The line count (2 bytes) indicates how many contiguous unused lines there are in the library file, starting with the next record in the library file.

The next_free pointer (2 bytes) is the record number of the next record in the free-space management list, this is used when the librarian is searching for a free block with enough space to add a new object file to the library. If the next_free pointer is zero, there are no more records in the free space list.

An object code section in the library file has the exact same format as a normal object file, it must begin with a >01 or "0" tag, and end with a record whose first character is ":". After finding a symbol in the library file, GenLINK loads the indicated object section as if it were a standalone object file.

Image file format:

The headers for the various "types" of images produced by GenLINK have the following formats:

0 "P" "F" <length> <load address>
max length = >3DFA.

1 >00 "F" <length> <load address>
max length = >3DFA.

2 "P" "G" <length> <load address>
max length = >3DFA.

3 >00 "G" <length> <load address>
max length = >3DFA.

4 >FF >FF <length> <load address>
max length = >1FFA.

5 >00 >00 <length> <load address>
max length = >1FFA.

6 >FF >01..>08 <length> <load address>
max length = >2000. The 2nd byte is
(load address / 8192) + >01.

7 >00 >01..>08 <length> <load address>
max length = >2000. The 2nd byte is
(load address / 8192) + >01.

8 >FF >09 <length> <load address>
max length = >2000.

9 >00 >09 <length> <load address>
max length = >2000.

10 >FF >0A <length> <load address>
max length = >2000.

11 >00 >0A <length> <load address>
max length = >2000.

12 >FF >0B <length> <load address>
max length = >1000.

13 >00 >0B <length> <load address>
max length = >1000.

14 NO HEADER.

Debug file format:

lines with symbol records
null line
lines with file records
end of file

Symbol records in the debug file have the following format:

address,line #,file #,type,namlen(byte),symbol_name[namlen]

File records in the debug file have the following format:

file #,namlen(byte),file_name[namlen]

Example control file (MDOS image):

```
*
* linker control file for Picture Transfer
*
ADD PIC
ADD UTIL
ADD USE
ADD SETSCR
ADD DISPLAY
ADD GSHOW
ADD GSAVE
ADD SETWIN
ADD MSAVE
*
STATUS
*
SAVEALL PICT
*
EXIT
*
```

Example control file (4A image):

```
*
* linker control file for  Fast-Term
*
* Fast-Term only loads in high 24k bank
*
BLOCK >A000,>FFD8
*
* load it
*
ADD FAST-TERMO
*
* check memory usage
*
STATUS
*
* display the length of Fast-Term
*
EVAL SLAST-SFIRST
*
* now to define a block which only contains
* relevant code (since Fast-Term loads some
* extra info after the end of the actual image)
*
CLEAR
BLOCK SFIRST,SLAST
*
* save it as a 4A image file...type 4
*
SAVEALL UTIL1,4
*
EXIT
*
```

Appendix A

LIB_4A is a library file containing subroutines and equates useful to programmers writing GPL (TI-99/4a) mode assembly language programs. These subroutines should be REFerenced from within your assembly language source programs.

These routines can be included in your program image files by GenLINK with the following procedure:

```
ADD     <your object files>
LIBREF  LIB_4A
SAVE    your_name,4
EXIT
```

The following pages describe the equates and subroutines which are defined by LIB_4A.

SYSTEM EQUATES

GPLWS	>83E0	workspace registers for GPL interpreter
GRMWA	>9C02	set address, port to GRAM/GROM chips
GRMRA	>9802	get address, port from GRAM/GROM chips
GRMWD	>9C00	set data, port to GRAM memory
GRMRD	>9800	get data, port from GRAM/GROM memory
PAD	>8300	address of high-speed memory in 4A console
SOUND	>8400	set data, port for sound chip
SPCHRD	>9000	get data, port from speech synthesizer
SPCHWD	>9400	set data, port to speech synthesizer
VDPWA	>8C02	set address, port to VDP chip
VDSTA	>8802	get status, port from VDP chip
VDPWD	>8C00	set data, port to VDP ram.
VDPRD	>8800	get data, port from VDP ram.
SCAN	>000E	address of built-in key scan routine.

UTLREG

This is a 32 byte area, initialized to zero in LIB_4A, which is used as registers in the subroutines VSBR, VSBW, VMBR, VMBW, VWTR, GPLLNK, XMLLNK, KSCAN and KSCANC.

You can override the DEFinition in LIB_4A by DEFining "UTLREG" within your own object modules. The most common reason for overriding the default in LIB_4A is to force the registers to locate in PAD ram, to increase performance of the VDP access routines.

VDP ACCESS SUBROUTINES

You may leave interrupts on while you call any of the following VDP access subroutines. The routines turn off interrupts as needed, then restore your interrupt mask before they return to you.

All of VDP access routines REFerence "UTLREG" as an external symbol. "UTLREG" is defined in LIB_4A as a block of 32 bytes initialized to zero. For performance reasons, you may want to override the definition of "UTLREG" provided in LIB_4A by DEFining "UTLREG" within one of of your object modules (you would generally DEFine it to be in PAD ram if you wanted better performance.)

VSBR	read single byte from VDP ram
VSBW	write single byte to VDP ram
VMBR	read multiple bytes from VDP ram
VMBW	write multiple bytes to VDP ram
VWTR	write to VDP register

The preceding routines perform the functions described in the TI Editor Assembler manual.

In addition to the preceding entry points, LIB_4A defines another routine, "VSETAD" which should not be REFerenced or DEFined within your code.

GPLLNK

LIB_4A defines a GPLLNK subroutine which functions as described in the TI Editor Assembler manual. It can be called from any program image, and it will restore the GROMs to their original state before returning to you.

XMLLNK

LIB_4A defines an XMLLNK subroutine which functions as described in the TI Editor Assembler manual, with the exception that the CIF (Convert Integer to Floating point) routine is not provided as an XML subroutine.

KSCAN

KSCAN functions as described in the TI Editor Assembler manual.

KSCANC

KSCANC is a key scan routine unique to LIB 4A. On a TI-99/4A console, it provides you with a "CRU" key scan routine which can be called while interrupts are enabled (KSCANC will not function on a GENEVE computer.)

You call KSCANC exactly as you would KSCAN. KSCANC does not have the full flexibility of KSCAN, only one scan mode is provided, and only codes defined in the following table will be returned.

Column 1: normal keycode.
Column 2: keycode when "shift" is also pressed.
Column 3: keycode when "control" is also pressed.
Column 4: keycode when "function" is also pressed.
Column 5: keycode when "function" and "shift" are also pressed.

Keycap	1	2	3	4	5
1	'1'	'!'	>B1	>83	>F3
2	'2'	'@'	>B2	>84	>D3
3	'3'	'#'	>B3	>87	>DB
4	'4'	'$'	>B4	>82	>E3
5	'5'	'%'	>B5	>8E	>EB
6	'6'	'^'	>B6	>8C	>EC
7	'7'	'&'	>B7	>81	>E4
8	'8'	'*'	>1E	>86	>DC
9	'9'	'('	>1F	>8F	>D4
0	'0'	')'	>B0	>BC	>F4
=	'='	'+'	>85	>85	>85

Q	'q'	'Q'	>11	>C5	>F1
W	'w'	'W'	>17	'~'	>D1
E	'e'	'E'	>05	>8B	>D9
R	'r'	'R'	>12	'['	>E1
T	't'	'T'	>14	']'	>E9
Y	'y'	'Y'	>19	>C6	>ED
U	'u'	'U'	>15	'_'	>E5
I	'i'	'I'	>09	'?'	>DD
O	'o'	'O'	>0F	'"'	>D5
P	'p'	'P'	>10	'`'	>F5
/	'/'	'-'	>BB	>8A	>F7
A	'a'	'A'	>01	'|'	>F3
S	's'	'S'	>13	>88	>D3
D	'd'	'D'	>04	>89	>DB
F	'f'	'F'	>06	'{'	>E3
G	'g'	'G'	>07	'}'	>EB
H	'h'	'H'	>08	>BF	>EE
J	'j'	'J'	>0A	>C0	>E6
K	'k'	'K'	>0B	>C1	>DE
L	'l'	'L'	>0C	>C2	>D6
;	';'	':'	>1C	>8D	>F6
<ENTER>	>0D	>0D	>0D	>0D	>0D
Z	'z'	'Z'	>1A	'\'	>F0
X	'x'	'X'	>18	>8A	>D0
C	'c'	'C'	>03	'@'	>D8
V	'v'	'V'	>16	>7F	>E0
B	'b'	'B'	>02	>BE	>E8
N	'n'	'N'	>0E	>C3	>EF
M	'm'	'M'	>0D	>C4	>E7
,	','	'<'	>00	>B8	>DF
<SPACE>	>20	>20	>20	>20	>20

ALPHA-LOCK causes keycodes in the range 'a'..'z' to be converted to the range 'A'..'Z'.

Like KSCAN, KSCANC returns the keycode at >8375, and returns the status flag at >837C if a new key has been pressed.

DEVICE DRIVER ACCESS ROUTINES

DSRLNK

DSRLNK functions as described the TI Editor Assembler manual.

DSRGO

DSRGO is a routine to similar in function to DSRLNK, but enhances performance of your programs. Each time you call DSRLNK, it searchs all of the ROMs in order until it finds the device you specified in your PAB. This search on every call to DSRLNK can add a lot of extra time to the i/o routines in your program, slowing your program down. To use DSRGO, you must call DSRLNK once, to OPEN the device, then save some variables which DSRLNK provides for use with DSRGO. On subsequent device access you would call DSRGO, with a pointer to the saved variables from the initial DSRLNK, and avoid the extra search on all device access subsequent to the DSRLNK.

DSRLNK provides the following variables for use with DSRGO:

DSRCRU	cru address of peripheral
DSREND	pointer to end of device name in PAB
DSRENT	address of entry point to DSR ROM
DSRLEN	length of device name in PAB
DSRSTA	pointer to status byte in PAB
DSRVER	version number of DSR routine at entry point
DSRREG	register block shared by DSRGO and DSRLNK

None of the previous symbols should be DEFined in your programs if you plan to use DSRLNK or DSRGO.

```
*
* usage example of DSRLNK and DSRGO, including definition of
* "save" area for DSRGO parameters.
*
            REF   DSRLNK,DSRGO
            REF   DSRCRU,DSREND,DSRENT
            REF   DSRLEN,DSRSTA,DSRVER
            REF   VMBW,VSBW
*
* open the device in the PAB
*
            LI    R0,VDPPAB       address of PAB in VDP ram
```

```
            LI      R1,NEWPAB       initial PAB in CPU ram
            LI      R2,PABLEN       length of PAB
            BLWP    @VMBW           copy PAB to VDP ram
*
            AI      R0,9            offset of name in PAB
            MOV     R0,@>8356       for DSRLNK
*
            BLWP    @DSRLNK
            DATA    8    normal device access
*
            JEQ     ERROR           i/o error
*
* save the DSRGO variables in the proper order
*
            MOV     @DSRCRU,@PABSAV+0
            MOV     @DSRENT,@PABSAV+2
            MOV     @DSRLEN,@PABSAV+4
            MOV     @DSREND,@PABSAV+6
            MOV     @DSRVER,@PABSAV+8
            MOV     @DSRSTA,@PABSAV+10
*
* on all subsequent access for this PAB, we can now use DSRGO
* as follows:
*
            LI      R0,VDPPAB       address of PAB opcode in VDP
            LI      R1,>0200        READ opcode in high byte
            BLWP    @VSBW           put the READ opcode into the PAB
*
            BLWP    @DSRGO
            DATA    PABSAV          pointer to saved variables
            JEQ     ERROR           i/o error
*
* ...       more of your code, using DSRGO with READs and WRITEs can
*            easily enhance your programs by a factor of two
* ...
*
PABSAV      BSS     12              6 variables to save
*
NEWPAB      DATA    >0014,>0000
            DATA    >5050,>0000
            DATA    NAMLEN
PABNAM      TEXT    'DSK1.EXAMPLE'
NAMLEN      EQU     $-PABNAM
PABLEN      EQU     $-NEWPAB
*
            END
*
```

Appendix B

LIB_MD is a library file containing subroutines and equates useful to programmers writing MDOS mode assembly language programs. These subroutines should be REFerenced from within your assembly language source programs.

These routines can be included in your program image files by GenLINK with the following procedure:

```
ADD     <your object files>
LIBREF  LIB_MD
SAVE    your_name,4
EXIT
```

The following pages describe the equates and subroutines which are defined by LIB_MD.

GETENV

Translate a character list in your task header into a useful string. The resulting string will be preceded by a length byte, and is terminated with a NULL (>00) byte. The pointer you pass to GETENV for the string is the address of the first character in the string, immediately after the length byte. See task information in the GenREF manual for information on which character lists are predefined for your use.

```
        LI      R0,HEADER           pointer to head of environment string
        LI      R1,STRING           address of buffer to put string into
        MOVB    @MAXLEN,@-1(R1)     length of buffer
        BL      @GETENV             uses R0 to R6
*
* ...
*
STRLEN  DATA    0
STRING  BSS     BUFLEN
        BYTE    >00
MAXLEN  BYTE    MAXLEN
```

SETENV

Translate a string in your program to a character list in your task's header. Useful to change drive assignments (used by the PARSE filename function of MDOS), current directories on block devices (also used by the PARSE filename function of MDOS), and to pass a command string to a spawned task. See task information in the GenREF manual for information on character lists which are used by MDOS functions. The string you pass to this subroutine must be preceded by a length byte, and must be terminated with a NULL (>00) character. If the string you pass is shorter than the list which is currently in use by MDOS, the list will be truncated and the extra nodes in the list will be added to the free-list for later use by MDOS.

```
            LI      R0,HEADER   pointer to head of environment string
            LI      R1,STRING   address of string to put into char list
            BL      @SETENV     uses R0 to R6
*
* ...
*
STRLEN      DATA    LENGTH
STRING      TEXT    'This is an example'
            BYTE    >00
LENGTH      EQU     $-STRING
```

EMIT (subroutine)

EMITF (printer flag)

The EMIT subroutine writes a single character to the screen using the MDOS WRITETTY video function, it will also echo the character to the printer if EMITF is zero and the control-P flag in your task header is set (see the GenREF manual for more information on the control-P and control-S flags.) EMIT will suspend execution of your program if the control-S flag in your task header is set (you program will be restarted when the control-S flag is cleared by MDOS.)

The character to be written to the screen must be in the high byte of your R0.

EMIT uses a workspace at >F060, the values of registers R0-R2,R13-R15 in this workspace are altered by use of EMIT. (make sure that calling EMIT doesn't alter any values your program needs to have saved.)

```
LI      R0,>1A00      clear screen code, in high byte
BLWP    @EMIT
```

PRINT (subroutine)

The PRINT subroutine writes a NULL terminated string to the screen using the MDOS WRITETTY video function, it will also echo the string to the printer if the control-P flag in your task header is set (see the GenREF manual for more information on the control-P and control-S flags.) PRINTF will suspend execution of your program if the control-S flag in your task header is set (you program will be restarted when the control-S flag is cleared by MDOS.)

The string to be written to the screen must immediately follow the call to PRINT.

PRINT uses a workspace at >F060, the values of registers R0-R2,R13-R15 in this workspace are altered by use of PRINT. (make sure that calling PRINT doesn't alter any values your program needs to have saved.)

```
        BLWP    @PRINT          send a string to screen
        TEXT    'Sample string'
        BYTE    0
*
* ...
*
```