DEFT Linker

1 Introduction 1

2 Operation 2
2.1 ORIGIN 2
2.2 LIST FILE: 3
2.3 BINARY FILE: 3
2.4 PASCAL? (Y) 3
2.5 DEBUGGER? (Y) 3
2.6 OBJ NAMES FILE: 4
2.7 OBJECT FILE: 4

3 Linker Map 5

4 Error Messages 8
4.1 BINARY FILE I/O ERROR 8
4.2 DUPLICATE - ... IN 8
4.3 DUPLICATE MAIN IGNORED 8
4.4 HEXWORDPARMMISSINGINOBJECTRECORD 8
4.5 INVALID DEBUG MODULE 8
4.6 INVALID MARKER 8
4.7 INVALID OBJECT RECORD 9
4.8 MODULE TOO BIG 9
4.9 NO MAIN ENTRY 9
4.10 OBJECT FILE I/O ERROR 9
4.11 PHASE ERROR 9
4.12 SYMBOL MISSING IN OBJECT RECORD 9
4.13 SYMBOL TABLE FULL - ... IN 9
4.14 UNDEFINED - ... IN 9

5 Limitations 10

Link

1 Introduction

DEFT Linker is a program which reads the object files produced by the **DEFT Macro/6809** Assembler or **DEFT Pascal** Compiler and produces an executable binary image suitable for loading with *Disk Extended Basic's* LOADM command. **DEFT Linker** features the following facilities:

- Object code relocation
- Automatic Pascal runtime modules inclusion
- Builtin **DEFT Debugger** interface
- Support for object module libraries. Object module libraries constructed by **DEFT LIB**, consisting of many object module files can be specified as input to **DEFT Linker**. Only those library sections referenced by your program will be included in the resulting binary.
- Multiple object file input, either explicit or via a separate ASCII file.
- *Disk Extended Basic* compatible binary output file.

2 Operation

Once you have created the necessary object files with the compiler and assembler, you are ready to link them together into your final binary image. The command *LOADM "LINKER":EXEC* will load **DEFT Linker** from disk drive 0 and begin execution.

DEFT Linker operates in three phases. During the first phase it displays the **DEFT Linker** screen and prompts you for the information required in subsequent phases.

The second phase starts after all the prompting is completed. During this phase it reads the object files, builds its symbol table of *public* symbols (relocating those symbols that need it), prints the module by module portion of its list file and reports any errors found in the object files.

The third phase involves **DEFT Linker** once again reading all the object files. On this last phase it performs all necessary relocation, fixups and *external* reference resolution while creating the final binary image. At the end of this phase **DEFT Linker** prints the symbol table.

The following provides an explanation of each prompt made by **DEFT Linker.**

2.1 ORIGIN

This is the decimal memory address where the resulting binary image is to be loaded by the *LOADM* command. For non-position independent files, this is the position from which the binary *must* execute. If the resulting image is position independent then a parameter can be added to the LOADM command to load the resulting file at a higher memory address.

If no origin is specified, then it defaults to 5000 (decimal). When you *PCLEAR 1, FILES 0,0* and *CLEAR 16,4999*, the 4999 of the last command tells BASIC that 4999 (decimal) is the highest memory location that BASIC is allowed to use. Therefore the lowest memory location available for your use starts at 5000 (decimal). From this memory location on up is now available for your specific use. This then, 5000 (decimal), becomes the lowest memory address which is protected from BASIC.

If you wish to write programs that are called from BASIC programs, then you will have to determine how much memory BASIC will need and enter an ORIGIN which is high enough to

provide that much memory.

2.2 LIST FILE:

This is the standard file name (with a default suffix of LST) of a file to be created by **DEFT Linker** which reports the results of the link. **DEFT Linker** will not produce any file if no file name is entered for this prompt.

2.3 BINARY FILE:

This is the standard file name (with a default suffix of BIN) of a disk file to be created by the **DEFT Linker.** This file name must be given and it must be a disk file.

2.4 PASCAL? (Y)

This prompt requires a *Y* or *N* response. Actually, any response other than *N* or *n* (including no response) is interpreted as yes. When this question is answered yes, the Pascal boot module *(PASBOOT/OBJ)* and runtime library *(RUNTIME/LIB)* are included. Only those segments of the runtime library referenced by your program will be included in the resulting binary load module. This means that the resulting program will be no larger than it has to be. Unused Pascal runtime features will not be included.

RUNTIME/LIB and *PASBOOT/OBJ* must both be present on disk drive 0.

2.5 DEBUGGER? (Y)

Like the *PASCAL?* question, the assumed answer is yes unless an *N* or *n* is entered. When this is answered affirmatively, the module DEBUGGER/LIB:0 is included in the binary. In addition, any Pascal modules which were compiled with the *debug* option turned on will have breakpoints generated and a module table will be included for use by the debugger.

If this question is answered negatively, then **DEFT Debugger** is not included, Pascal modules with the *debug* option turned on will have NOPs generated in place of breakpoints and no module table will be produced.

NOTE: if you have the **DEFT Pascal Workbench** and answer the *PASCAL?* question NO and the *DEBUGGER?* question YES, then

you will have to enter *RUNTIME/LIB* as one of the object files in either your OBJ NAMES FILE or to one of the OBJECT FILE prompts. This is because **DEFT Debugger** uses some of the facilities in the Pascal runtime library. If you have only **DEFT Bench**, then you do not have to do this since everything is included in the DEBUGGER/LIB library.

2.6 OBJ NAMES FILE:

When a large program has been divided into a number of modules, it is sometimes convenient to create a text file with the editor that lists the names of the object files to be included so that you don't have to individually type them in each time you link the program. This prompt allows you to specify the name of such a file.

This file must have 1 standard file name per line. The default suffix for the file names included in the file is OBJ. The default suffix for the OBJ NAMES FILE itself is LNK. When you enter a file name for this prompt, **DEFT Linker** does not prompt you for individual object file names.

2.7 OBJECT FILE:

This prompt is made if you did not provide an OBJ NAMES FILE. You provide a *single* object file name. **DEFT Linker** will verify that it can open the file and then prompt you for another file name. If more than one object file is to be included, enter the additional object file names one at each prompt. Once you have entered all the names, just hit the *ENTER* key on the last prompt and **DEFT Linker** will begin its second phase.

3 Linker Map

The following is a brief description of the Linker Map listing produced by **DEFT Linker** during linking operations.

1. *Header* - This is the first line of every page of the linker listing. The Header includes the page number.

2. *Module Name* - Every object file or module linked in a linker operation is identified by object file name. Proceding each module name, the following is printed:

   - *Object Generator* - This first line following the object file name identifies the compiler or assembler that produced the object file.

   - *Title(s)* - All titles produced within a program source file, with the title directives for both the compiler and assembler, are printed following the object generator identification. If a program contains no title(s) then none are printed.

   - *MODULE ORIGIN* - The four digit number following this title is the hexadecimal representation of the address in memory where that module will begin within the program.

   - *MODULE SIZE* - The four digit number following this title is the hexadecimal representation of the the number of bytes in memory that this module requires.

3. *Symbol Table* - At the end of every linker operation a symbol table is produced. Printed under this heading are the names of the symbols referenced by that program. Each element of this table is as follows:

   - *Symbol Value* - This is a four digit number which precedes every symbol table entry. This four digit number is a hexadecimal representation of the value or program address which the symbol is used to reference.

   - *Symbol Type* - This is the one or two character field which immediately follows the symbol value. This field identifies whether a symbol represents an absolute value (A), a program relative value (R), or a duplicate reference (D).

   - *Symbol Name* - this field immediately follows the symbol type. The symbol name is the string of characters used to reference a program value.

4. *Position Independence* - This is the seventh from the last line printed on the linker map listing. The character expression found on this line indicates whether the linked program is position idependent or non-position independent. PIC indicates that the resulting machine program contained in the program's load module file is Position Independent Code.

5. *ORIGIN* - The four digit number following this title is the hexadecimal representation of the address in memory where this program begins.

6. *LAST ADDR* - The four digit number following this title is the hexadecimal representation of the last address in memory where this program resides.

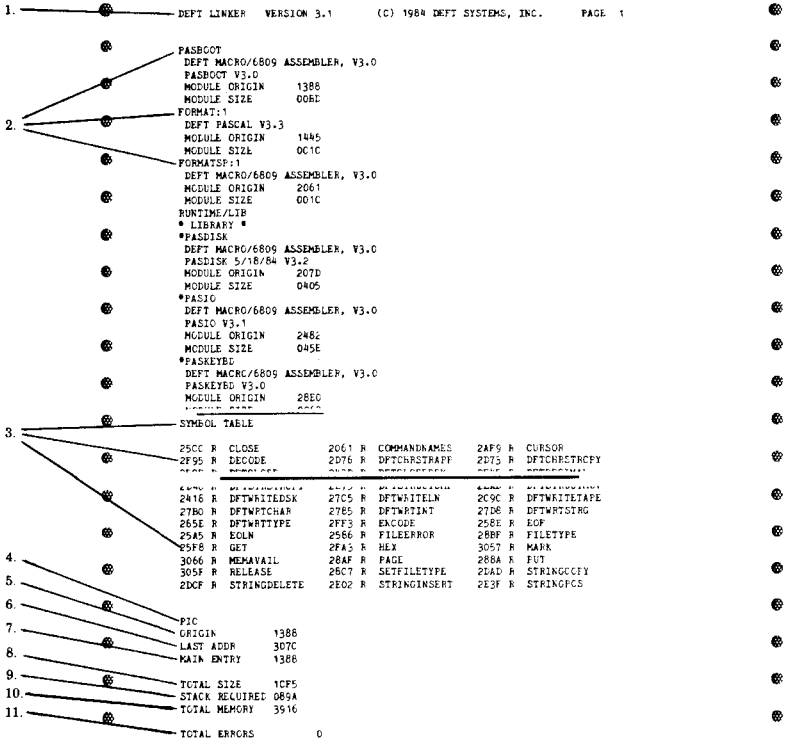
7. *MAIN ENTRY* - The four digit number following this title is the hexadecimal representation of the first address in memory where this program begins its execution.

8. *TOTAL SIZE* - The four digit number following this title is the hexadecimal representation of the total number of bytes of memory required to hold the program's executable instructions.

9. *STACK REQUIRED* - The four digit number following this title is the hexadecimal representation of the worst case number of bytes of stack memory required to execute the resulting machine program. It is the sum of the stack requirements of each individual module.

10. *TOTAL MEMORY* - this is the next to the last line printed on the linker map listing. The four digit number following this title is the hexadecimal representation of the total number of bytes of memory required to execute the resulting machine program.

11. *TOTAL ERRORS* - This is the last line printed on the linker map listing and is the number of errors encountered by **DEFT Linker** during its execution.




```
DEFT LINKER   VERSION 3.1      (C) 1984 DEFT SYSTEMS, INC.      PAGE  1

PASBOOT
 DEFT MACRO/6809 ASSEMBLER, V3.0
 PASBOOT V3.0
 MODULE ORIGIN      1388
 MODULE SIZE        00BE
FORMAT:1
 DEFT PASCAL V3.3
 MODULE ORIGIN      1445
 MODULE SIZE        0C1C
FORMATSP:1
 DEFT MACRO/6809 ASSEMBLER, V3.0
 MODULE ORIGIN      2061
 MODULE SIZE        001C
RUNTIME/LIB
* LIBRARY *
*PASDISK
 DEFT MACRO/6809 ASSEMBLER, V3.0
 PASDISK 5/18/84 V3.2
 MODULE ORIGIN      207D
 MODULE SIZE        0405
*PASIO
 DEFT MACRO/6809 ASSEMBLER, V3.0
 PASIO V3.1
 MODULE ORIGIN      2482
 MODULE SIZE        045E
*PASKEYBD
 DEFT MACRO/6809 ASSEMBLER, V3.0
 PASKEYBD V3.0
 MODULE ORIGIN      28E0

SYMBOL TABLE

25CC R  CLOSE           2061 R  COMMANDNAMES    2AF9 R  CURSOR
2F95 R  DECODE          2D76 R  DFTCHRSTRAPP    2D75 R  DFTCHRSTRCPY
2416 R  DFTWRITEDSK     27C5 R  DFTWRITELN      2C9C R  DFTWRITETAPE
27B0 R  DFTWPTCHAR      2785 R  DFTWRTINT       27D8 R  DFTWRTSTRG
265E R  DFTWRTTYPE      2FF3 R  EXCODE          258E R  EOF
25A5 R  EOLN            2586 R  FILEERROR       28BF R  FILETYPE
25F8 R  GET             2FA5 R  HEX             3057 R  MARK
3066 R  MEMAVAIL        28AF R  PAGE            288A R  PUT
305F R  RELEASE         28C7 R  SETFILETYPE     2DAD R  STRINGCOPY
2DCF R  STRINGDELETE    2E02 R  STRINGINSERT    2E3F R  STRINGPOS

PIC
ORIGIN          1388
LAST ADDR       307C
MAIN ENTRY      1388

TOTAL SIZE      1CF5
STACK REQUIRED  089A
TOTAL MEMORY    3916

TOTAL ERRORS          0
```

4 Error Messages

The **DEFT Linker** generates error messages during its second phase. These messages usually involve duplicate or missing public variable definitions. The error messages start with "***" and are as follows:

4.1 BINARY FILE I/O ERROR

An I/O error was detected while attempting to write to the binary output file. This could be caused by a full disk or the write protect being left on the diskette.

4.2 DUPLICATE - ... IN ...

The specified public symbol being defined in the specified object file has already been defined.

4.3 DUPLICATE MAIN IGNORED

More than one main object module has been found, any main modules found after the first one will be assumed to be a non-main module. There can be only one place in the program where execution is to start, that is in the main module.

4.4 HEX WORD PARM MISSING IN OBJECT RECORD

An invalid format object record has been detected. This may be due to the wrong type of file being input to the Linker.

4.5 INVALID DEBUG MODULE

The necessary public symbols have not been defined when the *DEBUGGER?* question has been answered with yes. This is probably due to not having the file *DEBUGGER/LIB* present on drive 0 while linking.

4.6 INVALID MARKER

An invalid format language marker record has been found in the object file. This may be due to the wrong type of file being input to the Linker.

4.7 INVALID OBJECT RECORD

An invalid format object file record has been found. This may be due to the wrong type of file being input to the Linker.

4.8 MODULE TOO BIG

The module being processed is too big to be processed by the Linker.

4.9 NO MAIN ENTRY

No main module has been included. The entry point is assumed to be the beginning of the binary image.

4.10 OBJECT FILE I/O ERROR

An I/O error was detected while attempting to read an object file. This error also occurs if you don't have *RUNTIME/LIB* or *PASBOOT/OBJ* on drive 0 when linking a Pascal program.

4.11 PHASE ERROR

The value of a symbol is different in the Linker's second and third phases. This error should not occur and indicates some fundamental problem with either the Linker or the object files.

4.12 SYMBOL MISSING IN OBJECT RECORD

An invalid format object record has been detected. This may be due to the wrong type of file being input to the Linker.

4.13 SYMBOL TABLE FULL - ... IN ...

The specified public symbol being defined in the specified object file cannot be put in the Linker's symbol table because it is full.

4.14 UNDEFINED - ... IN ...

The specified public symbol being referenced in the specified object file has not been defined.

5 Limitations

In addition to the above facilities, this version of **DEFT Linker** has the following limitations:

32K Memory Operation -

When running **DEFT Linker** in only 32K bytes of memory the following limitations apply:

1. A maximum of 50 object files can be linked together.
2. No object file can be larger than 4K bytes.
3. No more than a total of 400 public symbols can be defined in all the modules to be linked. The Pascal runtime package has about 80 in this version.

64K Memory Operation -

When running **DEFT Linker** in 64K bytes of memory the following limitations apply:

1. A maximum of 50 object files can be linked together.
2. No object file can be larger than 36K bytes.
3. No more than a total of 400 public symbols can be defined in all the modules to be linked. The Pascal runtime package has about 80 in this version.