DEFT Pascal Workbench

User's Guide

TRS-80™ Color Computer Software Series

Version 8 Second Printing

DEFT Pascal
DEFT Edit
DEFT Macro/6809
DEFT Linker
DEFT Debugger
DEFT Lib

Software License

DEFT Systems, Inc. grants to you, the customer, a non-exclusive, paid-up license to use the **DEFT Systems** software on **one** computer, subject to the following provisions:

1. Except as otherwise provided in the Software License, applicable copyright laws shall apply to the Software.
2. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to you, but not title to the Software.
3. You may use the Software on one host computer and access that Software through one or more terminals if the Software permits this function.
4. You shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in the Software License. You are expressly prohibited from disassembling the Software.
5. You are permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation **of one** computer with the Software, but only to the extent the Software allows a backup copy to be made.
6. You may resell or distribute unmodified copies of the Software provided you have purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from you.
7. All copyright notices shall be retained on all copies of the Software.

Term

This License is effective until terminated. You may terminate this License at any time by destroying the Software together with all copies in any form. It will also terminate if you fail to comply with any term or condition of the License.

Warranty

These programs, their instruction manual and reference materials are sold AS IS, without warranty as to their performance, merchantability, or fitness for any particular purpose. The entire risk as to the results and performance of these programs is assumed by you.

However, to the original purchaser only, **DEFT Systems, Inc.** warrants the magnetic diskette on which these programs are recorded to be free from defects in materials and faulty workmanship under normal use for a period of thirty days from the date of purchase. If during this thirty day period the diskette should become defective, it may be returned to **DEFT Systems, Inc.** for a replacement without charge, provided you have previously sent in your limited warranty registration notice to **DEFT Systems, Inc.** or send proof of purchase of these programs.

Your sole and exclusive remedy in the event of a defect is expressly limited to replacement of the diskette as provided above. If failure of a diskette has resulted from accident or abuse **DEFT Systems, Inc.** shall have no responsibility to replace the diskette under the terms of this limited warranty.

Any implied warranties relating to the diskette, including any implied warranties of merchantability and fitness for a particular purpose, are limited to a period of thirty days from the date of purchase. **DEFT Systems, Inc.** shall not be liable for indirect, special, or consequential damages resulting from the use of this product. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitations might not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Support

DEFT Systems, Inc. (and not Radio Shack) is completely responsible for the Warranty and all maintenance and support of the Software. Any questions concerning the Software should be directed to:

DEFT Systems, Inc.
P.O. Box 359
Damascus, Md. 20872

DEFT Pascal Workbench User's Guide

DEFT Pascal Workbench

1 DEFT Pascal Workbench

DEFT Pascal Workbench is a set of software development tools designed to support a programmer through the process of creating computer programs; from entering source code through executing the resulting machine program. **DEFT Pascal Workbench** is comprised of the following software packages:

DEFT Pascal
DEFT Edit
DEFT Macro/6809
DEFT Linker
DEFT Debugger
DEFT Lib

DEFT Pascal Workbench requires a TRS-80 Color Computer to be configured with at least 32K of memory, Extended Disk BASIC, and one floppy disk drive. **DEFT Pascal Workbench** utilizes a device independent file structure which is fully compatible with Disk Extended BASIC. Disk and tape files created with **DEFT Pascal Workbench** are of the same internal format as those produced and supported by BASIC.

1.1 DEFT Pascal

The **DEFT Pascal** Compiler is a fully recursive, single-pass Pascal language compiler for the TRS-80 Color Computer. It compiles Pascal programs directly into machine language code that can be executed by the 6809 microprocessor in the CoCo.

DEFT Pascal generally supports most standard Pascal language constructs. In addition, **DEFT Pascal** supports many extensions to the standard language which makes text processing, multi-language and systems type programs easier to write.

1.2 DEFT Edit

DEFT Edit is a screen mode, in-memory, text editor which provides its users with a selectively moveable *window* into a text file. **DEFT Edit** was designed primarily for the development of program source code, but it can also be used for the production of software documentation.

1.3 DEFT Macro/6809

DEFT Macro/6809 is a device-independent software package designed to translate Motorola 6809 Assembler source programs into 6809 micro-processor machine programs in two passes. Program source files may be read from either cassette or disk with the resulting machine program object files written to either cassette, disk, or the serial I/O port. **DEFT Macro/6809** parses and evaluates Motorola 6809 Assembler language statements and declarations, and generates the corresponding 6809 micro-processor machine programs according to Motorola 6809 Assembler language syntactical rules and conventions.

1.4 DEFT Linker

DEFT Linker is a program which reads the program object files produced by both **DEFT Pascal** and **DEFT Macro/6809** and converts them into machine executable binary image files suitable for loading with the Color Computer's LOADM command. **DEFT Linker** can also read multiple program object files and combine them into one larger machine executable binary *Load Module* so as to allow Color Computer users to develop very large programs one piece at a time.

1.5 DEFT Debugger

DEFT Debugger is an excellent tool for debugging machine programs developed in either Pascal or Assembler. **DEFT Debugger** allows you to stop and start a program under test at almost any point. Once the program under test has been stopped, you can display and/or change any memory location or micro-processor register.

When used with **DEFT Pascal**, **DEFT Debugger** provides symbolic access to your program as well as a trace facility for displaying currently active procedures.

1.6 DEFT Lib

DEFT Lib is an excellent tool for the development of object module libraries using object modules produced by either **DEFT Pascal** or **DEFT Macro/6809**. **DEFT Lib** is a device independent software package capable of creating and maintaining up to 50 object module sections in one library file. Once created, these libraries can be used

as input to **DEFT Linker** which will only use those sections which have been referenced by the particular program which is being linked.

2 DEFT Pascal Workbench Users Guide

The **DEFT Pascal Workbench** Users Guide is structured to be helpful in understanding and using **DEFT Pascal Workbench.** The Users Guide is not intended to be a self teaching guide in how to program but rather a tutorial on how to use the programs in **DEFT Pascal Workbench.**

If you already have an understanding of programming, then the User's Guide should contain more than enough information for you to immediately begin programming. If you have only programmed in BASIC, then you should be able to begin programming but you may need a Pascal text book when tackling some of the more advanced portions of the language. In either case, practice makes perfect, and no one should expect too much of themselves without some experience.

2.1 Document Divisions

The **DEFT Pascal Workbench** User's Guide is presented in three parts: *Introduction*, *How To* and *Background*. Each section was written with two specific objectives in mind.

- To support **DEFT Pascal Workbench** users according to their operation of a **DEFT** software product.
- To provide background information for reference.

2.2 Document Section Descriptions

The *Introduction* section informs the reader of two things. First, it describes the contents of the User's Guide itself and second, it describes how, in general terms, to use **DEFT Pascal Workbench** to develop programs.

The *How To* section describes in operational detail how to execute each tool provided in **DEFT Pascal Workbench.** This section starts with a **Familiarization Exercise** designed to be performed by you when you are first becoming acquainted with **DEFT Pascal.** This exercise provides a working example program. Following the exercise are individual sections which describe the operation and use of each program in the **DEFT Pascal Workbench.**

The *Background* section presents the reader with reference information. The first part summarizes the standard language elements of **DEFT Pascal** and includes a brief explanation of each. The second part summarizes the language extensions that are

contained in **DEFT Pascal**, with an explanation of each element. The last part summarizes the language elements of **DEFT Macro/6809** assembly language.

Regardless of how much experience a you may have, we highly recommend that you read the entire User's Guide. Good luck and have fun with **DEFT Pascal Workbench.**

3 Software Development

Developing programs with the **DEFT Pascal Workbench** is somewhat different from the procedure for developing programs in BASIC. With BASIC, you essentially type in the program and then type RUN. Debugging usually consists of hitting the BREAK key at appropriate points, PRINTing variables and turning the trace on and off.

This is a very good environment in which to develop small programs which do not have to execute with exceptional speed. However, as the programs you write become larger and more complex, some of the limitations imposed by the BASIC language will come in to play. These are primarily the small identifier size, lack of program structure, and execution performance of the interpreter.

DEFT Pascal Workbench takes up where BASIC leaves off. It should be seen as a powerful addition to your existing program tools. It is ideal for those programs which become very large, complex, and which execute for relatively long periods of time. All the programs in the **DEFT Pascal Workbench** were themselves developed using the workbench.

In general, the **DEFT Pascal Workbench** allows you to divide and conquer a large problem in smaller pieces. The linkage facilities found in **DEFT Pascal** and **DEFT Macro/6809** provide a very simple and straightforward method for combining the program pieces. This linkage facility is an extra step in the program development process and for small programs may not provide many benefits. However, in larger programs, the ability to modularize and compile or assemble only a small piece of a program at a time can be invaluable.

Since you are producing 6809 micro-processor instructions with **DEFT Pascal**, you will be dealing directly with the CPU when you begin debugging your resulting machine language program. You will use the **DEFT Debugger** to perform this step.

3.1 Program Design Development

Design. This step is one that you consciously or unconsciously perform before typing in a program. At the very least you should:

- Decide exactly what things the program is supposed to do. These are the program's functions.

- Decide how to organize the program around these major functions. This will identify what your major program pieces are.
- Decide how each piece should be organized to perform its function.

For *very* large programs, you may want to go to even more detailed design before beginning your coding. Remember that organizing the program is half the job of solving the problem. This usually involves defining all of the major data elements that you will be using before writing the code that manipulates them.

3.2 Source Code Development

Edit. This familiar step is the entry of a program's instructions which usually begins about halfway through the design stage. At this point, you will be creating *source module files;* that is, each program that is entered is stored in its textual form in a file. This step is performed by the programmer using a text editor such as **DEFT Edit**. The resulting text file containing the program statements is referred to as a *source file* or *source module file.*

This step is very similar to that in BASIC, except that in BASIC once the program is entered, it can then be immediately executed by the BASIC interpreter. With **DEFT Pascal**, the program statements in text form must first be translated into machine instructions for execution by the 6809 micro-processor. This leads us to the next phase of program development.

3.3 Object Code Development

Compile/Assemble. This is a new step for those used to BASIC. This step involves transforming the *source module files* that you created with **DEFT Edit** into *object module files* which contain two things:

- The *machine language* version of your programs
- *Linkage* information that will allow one *object module file* to be combined with others

DEFT Pascal and **DEFT Macro/6809** are both used to perform this step. Both programs prompt the user for both the name of the *source module file* which it uses for input and the *object module file* which it produces.

3.4 Load Module Development

Link. This is the last step before actually executing your program. This step converts the previously created *object module files* into single *binary load module files.*

When **DEFT Pascal** creates its object module files, it includes calls to machine language routines in other object modules which were included on your **DEFT Pascal** diskette. These object modules are in a special file called a runtime library and provide services such as I/O, string and set handling as well as floating point arithmetic. All of these object modules must be combined together and all of the address references between these modules must be adjusted appropriately in order to create a working program.

DEFT Linker performs this whole operation. It prompts you for the name(s) of the object module file(s) to be linked, which it uses for input, and the name of the *load module file* which it produces. This step takes all of those object module files and combines them into a single file that can be loaded via the BASIC LOADM command.

3.5 Program Execution and Debugging

Execute/Debug. This step involves actually testing your program by providing it with test data developed during the design step to determine if the program is producing the correct results. The **DEFT Debugger** permits a programmer to stop and restart a program under test at any point within the program. The programmer may then examine any memory location and/or microprocessor register and change its contents if desired. With the **DEFT Debugger**, the user may specify up to eight program stopping or *break* points at one time.

DEFT Debugger is an object module that is linked into your program's load module by **DEFT Linker** and therefore becomes a part of it. It initially gains control when your program begins execution so that you can use it to control subsequent execution. Once your program is debugged, you can re-link it without the debugger which will make your program smaller and faster.

For most large programs, the first and last steps, design and debugging, take the majority of the total time spent on a program. In fact, in very large projects the first and last steps are broken into a number of sub-steps in order to keep the job to a manageable size.

4 Getting Started

This section of the **DEFT Pascal Workbench** User's Guide is meant to provide you with the operational details required to use **DEFT** software products on the TRS-80 Color Computer. This section is required reading before you should attempt anything with a **DEFT** software product.

4.1 Program Execution

All **DEFT** programs for the TRS-80 Color Computer are binary machine language programs that are loaded into memory with the LOADM command and executed with the EXEC command. Before executing any **DEFT** program or any program that you create with the **DEFT Pascal Workbench**, it is absolutely necessary to protect it from BASIC. This is done with the following set of 4 BASIC Monitor commands. These commands need to be entered only once, just before the first time that you load a **DEFT** program. Subsequent loads of **DEFT** software will not require the re-entry of these BASIC Monitor commands.

1. *NEW* - This command is not necessary if you have just turned on your Color Computer. It is used to initialize the memory area normally used by the BASIC Interpreter in the Color Computer's ROM.

2. *PCLEAR 1* - This command causes Extended BASIC to reserve the minimum number of 1.5K byte pages for graphics. Since no **DEFT** software product uses BASIC's graphics for presentation, this command releases otherwise unused memory for use by the program being loaded.

3. *FILES 0,0* - This command tells BASIC that you do not intend to access any disk files via BASIC. Note that even after executing this command you can still DIR, KILL and RENAME. However, you will not be able to COPY. Since each program of the **DEFT Pascal Workbench** is an independent machine program, none of the BASIC Interpreter's file facilities are required, thereby freeing up even more otherwise unused memory.

4. *CLEAR 16,4999* - This reserves the upper 59K (27K in a 32K system) bytes of memory for use by **DEFT** software products. It will leave a little over 300 bytes of memory for use by BASIC. This Color Computer BASIC Monitor directive must be entered exactly as presented in this example. The first directive argument, *16*, tells the BASIC Monitor how many bytes of

memory to reserve for BASIC strings. Since no **DEFT** software products use the Color Computer's BASIC language, 16 bytes of memory is more than enough. The comma (,) preceding this next number is required, the next number, *4999*, tells the BASIC Monitor the last or highest value "address" in memory that it is allowed to use. This number is expressed in decimal, thereby reserving the rest of the Color Computer's memory, from decimal address 5000 on up, for any **DEFT** software product.

It is absolutely essential that you perform these commands before executing any of the programs in the **DEFT Pascal Workbench.** If you do not, BASIC may "over-write" portions of any program that you may load. If that were to happen, the loaded program's execution will produce unpredictable results.

The BASIC command for executing any of the programs in the **DEFT Pascal Workbench** is *LOADM "<filename>":EXEC* and the possible filenames are:

PASCAL	**DEFT Pascal**
EDITOR	**DEFT Edit**
ASSEMBLE	**DEFT Macro/6809**
LINKER	**DEFT Linker**
LIB	**DEFT Lib**

4.2 64K Operation

Whenever any **DEFT** program first begins execution, it immediately changes the Color Computer's memory map to unmap the BASIC ROM and map in any RAM that may exist in the top 32K of memory. **DEFT** programs are all fully self-contained and so don't need the BASIC ROM to operate.

After changing the memory map, the program will check to see whether you have a 32K or 64K system and then adjust the size of its main data structure to whatever memory is available. The result of this is that these programs can access up to 64K bytes of memory in your Color Computer.

With **DEFT Pascal**, or any other **DEFT** high level language compiler, any programs that you create will be able to use all the available memory in the system for your data variables. The only restriction is that the program instructions (not stack) must fit in the lower 32K of memory since this is loaded via BASIC.

4.3 32K Operation

Some 32K systems may show the same RAM memory size as a 64K system. This will cause all programs to switch to memory map 1 which will cause the system to hang. If you have such a TRS-80 Color Computer, you will want to do the following:

1. Power on your Color Computer.
2. Make a backup of your distribution diskette and put the distribution diskette in a safe place.
3. Put the *un-write-protected copy* of the distribution diskette that you just made into drive 0.
4. Enter the 4 BASIC commands found in the *Program Execution* section.
5. Enter: *RUN"MAKE32K"* <enter>

The program will run for about a minute and after it finishes, the diskette in drive 0 will contain a 32K version of the software.

If you have a 64K system and want to write Pascal programs that access the BASIC ROMs, you can rename *PASBOOT/OBJ* to *PASBOOT/64K* and *PASBOOT/32K* to *PASBOOT/OBJ*. By doing only this, your **DEFT** software will still run using all 64K but any program linked using this new version of *PASBOOT/OBJ* will operate with the BASIC ROMs in place.

4.4 DEFT Files

One of the advantages of using the **DEFT Pascal Workbench** is the device independent file structure which is supported while remaining fully compatible with the TRS-80 Disk Extended Color BASIC System Software. Disk or tape files created with BASIC, **DEFT** software products or programs developed with **DEFT Pascal** are all of the same fundamental format.

When executing **DEFT** software development tools you will have to specify the names of the *source module*, *object module* and *binary load module* files. The file naming conventions used with the **DEFT Pascal Workbench** are only slightly different from that of BASIC and allow complete device independence. The format of the names are as follows:

<filename>/<ext>:<device#>

This is the same format that BASIC uses for Disk files. However, by extending the device numbers, **DEFT Pascal Workbench** also uses it for the keyboard, screen, tape and printer. The <filename> is 0 to 8 ASCII characters. The extension is 0 to 3 ASCII characters. The device numbers range from -3 to 3 with the following meanings:

-3 Keyboard/Screen
-2 Printer
-1 Cassette Tape
0 Disk drive 0
1 Disk drive 1
2 Disk drive 2
3 Disk drive 3

As can be seen, the positive device numbers correspond to BASIC's drive numbers. The negative device numbers correspond to BASIC's device numbers with the exception that the Keyboard/Screen is -3 rather than 0.

All of the fields are optional in different circumstances. When a device number of -3 or -2 is specified, there is no need for a <filename> or <extension>. When a device number of -1 is specified, the <extension> is not required. For device numbers 0 thru 3, a default <extension> is always present depending on the program being run. When a device number is not specified, 0 is assumed. Following are some examples:

:-3	**Keyboard/Screen**
:-2	**Printer**
MYFILE:-2	**Printer (filename ignored but allowed)**
TAPEFILE:-1	**Cassette Tape File**
DISKFILE/ASM	**Assembler source file on disk drive 0**
F2NAME:1	**File is on disk drive 1, default extension used**

4.5 DEFT Pascal Workbench Diskette Contents

The following files are contained on the diskette that you received. You are encouraged to make a copy of the distribution diskette for your own backup purposes and to execute from the backup rather than the original diskette.

1. *PASCAL/BIN* - This file contains the executable image of the **DEFT Pascal** Compiler.

2. *EDITOR/BIN* - This file contains the executable image of **DEFT Edit.**

3. *LINKER/BIN* - This file contains the executable image of the **DEFT Linker.**

4. *ASSEMBLE/BIN* - This file contains the executable image of **DEFT Macro/6809.**

5. *LIB/BIN* - This file contains the executable image of **DEFT Lib.**

6. *PASCALIB/EXT* - This is a Pascal source file which is automatically *copied* by **DEFT Pascal** at the beginning of all programs which it compiles. This file contains the declarations of all of the predefined procedures and functions provided with **DEFT Pascal.** This file must be present on disk drive 0 whenever **DEFT Pascal** is executed.

7. *PASBOOT/OBJ* - This is the object file for the standard *boot* code for all Pascal programs. All programs produced by **DEFT** Software have a *first instruction.* For **DEFT Pascal** programs these first instructions are kept in this file. This object module file contains the machine language routines for Pascal program initialization. This file must be present on disk drive 0 when linking a Pascal program with the **DEFT Linker.**

8. *RUNTIME/LIB* - This is the object module library file which contains all the Pascal Runtime routines for Pascal programs developed with **DEFT Pascal.** Each library section contains machine language routines which are automatically called by **DEFT Pascal** when you use various parts of the language. This file must be present on disk drive 0 when linking a Pascal program with **DEFT Linker.**

9. *DEBUGGER/LIB* - This is the library file which contains **DEFT Debugger** for debugging any program created with **DEFT Pascal Workbench.** This file must be present on disk drive 0 when linking any program which is to include **DEFT Debugger.** See **DEFT Debugger** for more information.

10. *FORMAT/PAS & FORMAT2/PAS* - These are the two source files which contain the *Text Formatter* **DEFT Pascal** program. You will use these source files in the *Familiarization Exercise* part of the HOW TO section, to create your own text processing system.

11. *FORMATSP/ASM* - This is a source file which contains the 6809 Macro Assembler language portion of the *Text Formatter* program.

12. *FORMATSP/OBJ* - This is an object file produced by **DEFT Macro/6809** from the *FORMATSP/ASM* source file. It is included on the distribution diskette in case you do not wish to use the assembler.

13. *FORMAT/TXT* - This is an ASCII file that the FORMAT program uses for input. The FORMAT program will produce a set of instructions describing how to use itself.

14. *PASBOOT/ASM* - This is a source file which contains 6809 Macro Assembler language instructions which are the very first instructions executed by any Pascal program developed via the **DEFT Pascal**.

15. *MAKE32K/PAS* This is a BASIC program that converts a distribution diskette to 32K operation.

4.6 Single Disk Drive Operation

When using a single disk drive system you will have to create a *work diskette* that contains a couple of files from the distribution diskette as well as your own source, object and binary files. To execute a program you will insert the distribution diskette into your disk drive, load the proper binary image, insert your work diskette into the drive and then execute the loaded program.

The files that need to be copied onto your work diskette are:

DEBUGGER/LIB
PASCALIB/EXT
PASBOOT/OBJ
RUNTIME/LIB

You can copy these files by using the COPY command in BASIC. Although single drive operation is not documented, this command works the same way BACKUP does in single drive mode.

On some early versions of Disk Extended Basic the COPY command will not work on a single disk drive. If you have one of these, use BACKUP to create a work diskette and then KILL all the files on the diskette except those named above.

Familiarization Exercise

..Exer

1 Introduction

In order to illustrate the use of the **DEFT Pascal Workbench**, a sample program has been included on the diskette. This program is made up primarily of a PASCAL program which is contained in the files *FORMAT/PAS* and *FORMAT2/PAS*. An assembler module *FORMATSP/ASM* contains a pre-initialized lookup table that is used by the Pascal program. The assembler module has already been assembled into an object file *(FORMATSP/OBJ)*, however, if you also have **DEFT Bench**, then you can also perform the section on assembling a program.

2 Design

This step has already been performed for you. The purpose of the program is to read an ASCII file, which can be created by **DEFT Edit**, and to produce a professional looking document. The input file for this program contains text and text processing commands which control how the resulting document is to look. Text processing commands are recognized by having a period (.) as the first character in a line. The document that will be produced as a result of this exercise, contains a *Detailed Functional Description* of what the program is to do.

The program is broken down into the following major procedures:

- *Initialize* initializes all variables and prompts for file names required.
- *ReadNextLine* reads the next line of input and determines whether it is a command or text. If it is a command, it determines which command that it is.
- *NextSymbol* parses an input command for each parameter of that command.
- *FillOutput* and *NoFillOutput* create normal output text from an input text line.
- One procedure per command will be used to process each command type.

3 Edit

This phase has also been performed. As mentioned before, the files *FORMAT/PAS* and *FORMAT2/PAS* contain the Pascal program. The file *FORMATSP/ASM* contains the assembly language support for the program. With **DEFT Edit** or your own ASCII file text editor, you can edit these files to see what they look like. We recommend that you don't make any changes to the program until after you have made a backup and have executed the final program at least once successfully.

4 Compile/Assemble

We are now ready to compile the Pascal program and assemble the assembler support code. This section assumes that you are using a two disk drive system with the **DEFT Pascal Workbench** diskette in drive 0 and your work diskette in drive 1.

If you have only a single drive system, then you will have to copy the following files onto your work diskette (see the section on *Single Drive Operation*):

FORMAT/PAS
FORMAT2/PAS
FORMATSP/ASM
FORMATSP/OBJ
FORMAT/TXT

Before starting make sure that you have performed the steps described under *Getting Started* to protect the machine language programs from BASIC.

4.1 Executing the DEFT Pascal Compiler

The command *LOADM "PASCAL":EXEC* will load the **DEFT Pascal** Compiler from disk drive 0 and begin execution. You will see the **DEFT Pascal** Compiler screen with all of its prompts. If you have only a single disk drive, then remove the **DEFT Pascal Workbench** diskette from the drive and insert your work diskette. Each prompt and its possible replies are described below:

- **SOURCE** requires the name of the source file which is to be compiled. The default extension is PAS. Your response for this sample program will be *FORMAT, FORMAT:0, FORMAT/PAS* or *FORMAT/PAS:0* all of which are equivalent.
- **OBJECT** requires the name of the object file that is to be created by the compiler. This can be either on tape or disk or the name can be ommitted entirely if you do not wish to create an object file. The default extension is OBJ. Your response for this sample program will be *FORMAT:1* or *FORMAT/OBJ:1* both of which are equivalent. If you have a single drive system, your response will be *FORMAT, FORMAT/OBJ* or *FORMAT/OBJ:0.*
- **LIST** requires the name of the list file which is to be created by the compiler. This can be tape, disk, screen or printer or the name can be ommitted entirely if you do not wish to create a list file. The default extension is LST. Your response for this sample program

will be :-2 if you have a printer or nothing if you don't.

- **DEBUG?** asks you whether you wish to have debug information included in the resulting object file. You can answer this either with *N*, *n* or anything else. Anything other than *N* or *n* (for No) is taken to be *Y* (for Yes).

  The debug information will make your program significantly bigger but will allow you to symbolically debug your resulting program if you answer the **DEFT Linker's** *debug?* question with a *Y*. If you specify *Y* to **DEFT Pascal's** *debug?* question and *N* to the **DEFT Linker's** debug question, then the debug information will still be in the final binary image even though the **DEFT Debugger** module is not present.

  If you want to try out the debugger, then you can answer this question *Y*, otherwise answer it *N*.

- **DIRECTIVE** requires any **DEFT Pascal** directive that you would like to include before any source lines are read. The section *Compiler Controls* describes all the possible compiler controls that you could enter here. Your response for this sample program will be *T<your name>* which will cause <your name> to be printed at the top of each page of the program listing.

After you answer the *DIRECTIVE* prompt, the program will begin executing. The compiler requires that the file PASCALIB/EXT be present on disk drive 0 at this point. When the compiler is finished executing, control will return to BASIC and you will get the OK prompt.

This execution of the **DEFT Pascal** Compiler will read both the FORMAT/PAS and FORMAT2/PAS source files and create the FORMAT/OBJ object file. The FORMAT2/PAS file will be read because of a compiler directive at the end of the FORMAT/PAS source file.

4.2 Executing the 6809 Macro Assembler

If you want to try out the **DEFT Macro/6809** assembler then you can also assemble *FORMATSP/ASM* into the *FORMATSP/OBJ* file. If you don't, then go to the next section.

First put the **DEFT Pascal Workbench** diskette in disk drive 0 and enter the command *LOADM "ASSEMBLE":EXEC* to load **DEFT Macro/6809** and begin its execution. If you have a single drive

system, put your work diskette into disk drive 0.

You will see the assembler's screen appear along with its first prompt. Each prompt and its possible replies are described below:

- **TITLE:** requires the string of characters that you want to see at the top of each page of your assembly listing. You do not have to enter a title but for this sample program you can enter your name.
- **SOURCE FILE:** requires the name of the source file which is to be assembled. The default extension is ASM. Your response for this sample program will be *FORMATSP*, *FORMATSP:0*, *FORMATSP/ASM* or *FORMATSP/ASM:0* all of which are equivalent.
- **OBJECT FILE:** requires the name of the object file that is to be created by the assembler. This can be either on tape or disk or the name can be ommitted entirely if you do not wish to create an object file. The default extension is OBJ. Your response for this sample program will be *FORMATSP:1* or *FORMATSP/OBJ:1* both of which are equivalent. If you have a single disk drive system, you response will be *FORMATSP*, *FORMATSP:0*, *FORMATSP/OBJ* or *FORMATSP/OBJ:0*.
- **LIST FILE:** requires the name of the list file which is to be created by the assembler. This can be tape, disk, screen or printer or the name can be ommitted entirely if you do not wish to create a list file. The default extension is LST. Your response for this sample program will be :-2 if you have a printer or nothing if you don't.

After you answer the *LIST FILE:* prompt, the assembler will begin its first pass. During this first pass only the disk will appear to be doing anything. For this sample program, the first pass should last only a few seconds. The assembler will begin printing on its second pass through the source code. During this second pass **DEFT Macro/6809** will read the *FORMATSP/ASM* source file and produce the *FORMATSP/OBJ* object file and a listing on your printer.

5 Link

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. Make sure that you have the **DEFT Pascal Workbench** diskette in disk drive 0 and then enter the command *LOADM "LINKER":EXEC* to load **DEFT Linker** and begin its execution. If you have a single drive system, put your work diskette in disk drive 0. The *Operation* section in the **DEFT Linker** documentation describes how to operate the Linker. For your sample program, the responses required will be:

- **ORIGIN** - no response, this will invoke the default origin.
- **LIST FILE:** - *;-2* if you have a printer, otherwise nothing.
- **BINARY FILE:** - *FORMAT:1* or *FORMAT/BIN:1* both of which are equivalent. If you have a single drive system, enter *FORMAT, FORMAT:0, FORMAT/BIN* or *FORMAT/BIN:0* all of which are equivalent.
- **PASCAL? (Y)** - Y.
- **DEBUGGER? (Y)** - *Y* if you want to try out **DEFT Debugger**, otherwise *N*.
- **OBJ NAMES FILE:** - no response, this is because you do not have a text file that contains the file names of all the object files to be linked.
- **OBJECT FILE:** - *FORMAT:1* or *FORMAT/OBJ:1* both of which are equivalent. If you have a single drive system, enter *FORMAT, FORMAT:0, FORMAT/OBJ* or *FORMAT/OBJ:0* all of which are equivalent.
- **OBJECT FILE:** - *FORMATSP:1* or *FORMATSP/OBJ:1* both of which are equivalent. If you have a single drive system, enter *FORMATSP, FORMATSP:0, FORMATSP/OBJ* or *FORMATSP/OBJ:0* all of which are equivalent.
- **OBJECT FILE:** - no response to indicate that you have entered all the object file names that you wish to link.

The Linker will then begin operation and produce both the final binary image in the file *FORMAT/BIN* and a listing on your printer.

6 Execute/Debug

The command *LOADM "FORMAT:1":EXEC (LOADM "FORMAT":EXEC* on a single drive system) will load the sample program and begin its execution. If you specified *Y* to the *DEBUGGER?* prompt from **DEFT Linker** then you will see the **DEFT Debugger** screen. The **DEFT Debugger** documentation provides a complete description of how to operate the debugger. If you did not specify *Y* or if you give **DEFT Debugger** the *GO* command, then you will see the *FORMAT* screen with its first prompt. You should answer the prompts as follows:

1. **INPUT FILE:** - FORMAT, FORMAT:0, FORMAT/TXT or FORMAT/TXT:0 all of which are equivalent.
2. **OUTPUT FILE:** -:-2 if you have a printer. If not, put the output on disk by entering FORMAT:1 or FORMAT/LST:1 both of which are equivalent. If you have a single disk system use *FORMAT, FORMAT:0, FORMAT/LST* or *FORMAT/LST:0* all of which are equivalent.

Once you answer the last prompt the program will begin executing and produce a document showing you how to use the program.

DEFT Edit

1 Introduction

DEFT Edit is a program that allows you to create and modify PASCAL and Assembler source programs as well as any type of ASCII text file. Its features include:

- Text is maintained in memory to provide excellent command response.
- Files can be read and merged from either cassette or disk. They may be written to cassette, disk or printer.
- The user interface is a screen-mode "window" into the text with automatic up/down and left/right scrolling.
- All keys are auto-repeat.
- The *FIND* command allows you to search for specific patterns, *CHANGE* provides for changing the pattern in 1 or more instances.
- *MARK* and *APPEND* commands allow copying and moving of portions of text to either other places in the working text or to a file.

2 Basic Operation

After *LOADMing* and *EXECing* **DEFT Edit** you will see **DEFT Edit**'s copyright screen which has the *INITIALIZE? (Y)* prompt. The editor uses the answer to this question to determine whether to initialize its in-memory text buffer. When you have just loaded the editor, you must answer this question yes. This can be done by entering anything other than *N* or *n* (including nothing) and then depressing the *ENTER* key. The only times that you would answer this question with a *N* or *n* is when you have previously used the editor, exited and did nothing to alter the computer's memory, and then re-entered **DEFT Edit**. See the *QUIT* command for more information.

Once the editor is loaded and initialized, you are now ready to enter text. The following sections will describe and explain what you see and what you can do.

2.1 Text Screen

Once you have answered the *INITALIZE? (Y)* question, you will see the text screen. This screen will be green with a blue square at the top left-hand corner and some numbers and letters on the bottom line in reverse video. The blue square is blinking and if you type some characters, they appear on the top line followed by a blinking orange square. The blue square has moved down to the second line. If you hold down a key, you see the corresponding character repeat. Each element on this screen is discussed in detail in the following subsections:

Blinking Square

There is always one square on the top 15 lines of the screen which blinks. This may be either a colored square, a character or a blank. The place on the screen which is blinking is the cursor. This is the point at which any text that you type in will appear. In addition, many commands that you can enter will affect text relative to the position of the cursor.

Blue Square

The blue square indicates the end of the text held in memory. Anytime the cursor is on a line which is within 14 lines of the end of the text, the blue square will appear at the left hand side of the screen on the line following the last line of text.

Orange Square

The orange square indicates the end of the line. It appears on the screen in the position that a carriage return is stored in memory. Every line, including the last line, always has a carriage return at the end.

Status Line

The line in reverse video at the bottom of the screen is the status line. This line provides information about the current status of your editting session. The information provided (in order) is:

1. The three characters at the left-hand side of the screen indicate the mode that the editor is in. INS (for insert) is the mode that the editor initially comes up in and causes each character typed to be inserted before the character pointed to by the cursor. The other modes are REP (for replace) and MARK which are discussed in later sections.
2. The number followed by the character *L* is the line number on which the cursor is currently positioned. The first line is numbered zero.
3. The number followed by the character *C* is the column number at which the cursor is currently positioned. The first column on a line is zero.
4. The number followed by the characters *LS* is the line size of the line on which the cursor is currently positioned. This count includes the carriage return at the end of the line.
5. The number followed by the character *T* is the number of remaining characters of text which can still be entered in memory. This number is updated each time the cursor is positioned to a new line.

Auto-Repeat

The auto-repeat feature allows you repeat the entry of any key on the keyboard by merely holding the key down for a full second. After this, the key will repeat at about 6 characters per second.

ENTER Key

The ENTER key is used to enter a carriage return into the text. This effectively splits the line at the cursor position and so creates two new lines.

SHIFT-0 Keys

The SHIFT-0 combination of keying, toggles the TRS-80 Color Computer from UPPER CASE into UPPER/lower case and from UPPER/lower case into UPPER CASE depending on what state the computer was in prior to the simultaneous entry of the SHIFT and 0 keys.

3 Cursor Positioning

As noted above, each character entered at the keyboard is displayed on the screen at the position of the cursor. The cursor then moves one column to the right. If the cursor is not currently positioned where you want it, you can use the four arrow keys to move the cursor. By depressing the appropriate up, down, left or right arrow key, the cursor will move in the same direction.

The cursor will always be positioned within the text of some line. This has the following side-effects:

1. When moving the cursor up or down, if the cursor moves from a long line to a short line such that it would be positioned beyond the end of the short line then the cursor will be positioned at the end of that line.
2. When moving the cursor to the right, if the cursor is at the end of the line then it will be positioned to the beginning of the next line.
3. When moving the cursor to the left, if the cursor is at the beginning of the line then it will be positioned to the end of the previous line.
4. When the cursor is positioned at the end of the text (blue square), the right and down arrows will not move it.
5. When the cursor is positioned at the beginning of the text (line 0, column 0) then the left and up arrows will not move it.

4 Scrolling

DEFT Edit lets you enter lines up to 255 characters long. This is considerably more than can be displayed on a 32 column by 15 line screen. The way that you view all of this text is by *scrolling* it past the screen. The screen becomes a window into the text.

This scrolling occurs automatically as you position the cursor by either entering text or by using the arrow keys. If the cursor is at the bottom of the screen and you force the cursor down to the next line, then all the lines on the screen move up 1 line with the top line disappearing and a new line appearing at the bottom of the screen. The reverse occurs when the cursor is positioned at the top of the screen and you force it to move up.

DEFT Edit also provides left and right scrolling in a similar manner. When the cursor is positioned at the rightmost column on the screen and you force it to move right, all the text on the screen shifts to the left by 12 columns. This prevents eye fatigue when entering data and having the text contantly scrolling to the left. The text will scroll to the right by 12 columns when the cursor is at the leftmost side of the screen and you force it to the left.

1 Functions

In addition to entering text, **DEFT Edit** provides many powerful functions that speed text editing. The general purpose functions are described in this section.

1.1 The CLEAR Key

The *CLEAR* key is used to invoke editor functions. When the *CLEAR* key is depressed, the cursor changes from a reverse video of the character that it is over to a white square. When the cursor changes to this white square, the next key entered is interpreted as a function rather than as a character to be entered into the text. Once the function is performed, the cursor returns to its normal reverse video state.

The *CLEAR* key itself becomes an *unCLEAR* function when it is depressed a second time, which returns the cursor to its normal mode without performing any function.

1.2 Major Cursor Positioning

By using the *CLEAR* key in conjunction with the arrow keys you can quickly position to a specific area of text. The CLEAR-arrow functions are as follows:

1. *CLEAR-Up Arrow* makes the cursor go UP by 15 lines to the beginning of that line. In addition, the line that the cursor is positioned to will be at the top of the screen.
2. *CLEAR-Down Arrow* makes the cursor go DOWN by 15 lines to the beginning of that line. In addition, the line that the cursor is positioned to will be at the top of the screen.
3. *CLEAR-Left Arrow* makes the cursor go to the beginning of the line that it is currently positioned on.
4. *CLEAR-Right Arrow* makes the cursor go the end of the line that it is currently positioned on.
5. *CLEAR-B* makes the cursor go to the beginning of the text.
6. *CLEAR E* makes the cursor go to 15 lines before the end of the text. This line is positioned at the top of the screen with the cursor at the beginning of the line. This allows you to see the last 15 lines in the text. This command may take a couple of seconds on large files due to counting carriage returns in the text in order to maintain the line number.

1.3 Up Arrow Character Entry

The *Up Arrow* character is used in Pascal to denote pointer and file dereferencing. It is also used for cursor positioning by **DEFT Edit**. By first typing the *CLEAR* key and then depressing the *SHIFT* key while typing the *Up Arrow*, the *Up Arrow* character will be entered into the text.

1.4 Deleting Characters

There are two ways of deleting characters. The first is with the *CLEAR-D* function. When you use this function the character that the cursor is positioned over is deleted and all the characters to the right of the cursor are shifted to the left one character.

If you delete the carriage return at the end of the line, the line following will be appended to the end of the line. You cannot delete the last carriage return in the text.

A second way to delete characters is with the shifted left arrow key. In this case the cursor is moved one position to the left and the character there is deleted as previously described.

A third way is to delete all characters from the position of the cursor (inclusive) to the end of the line. First you position the cursor over the first character in the line from where you wish to *hack-off* the rest of the line, then you enter *CLEAR-H*. This function will *hack* that section of the line away and delete those characters.

1.5 Deleting Lines

A complete line can be deleted by positioning the cursor to any character on the line to be deleted (including the carriage return) and entering *CLEAR-L*. This function allows you to delete the last carriage return (as well as the last line) in the text.

1.6 Replace/Insert Modes

When **DEFT Edit** is first executed, it is in the *insert* mode of text entry. In this mode, when a character is entered at the keyboard it is inserted in front of the character that the cursor is positioned over. A second mode that the editor can be placed in is the *replace* mode. In this mode, when a character is entered at the keyboard it replaces the character that the cursor is positioned over. However, if the cursor is positioned over a carriage return then the character is

inserted in front of it.

You can switch between these modes with the *CLEAR-I* and *CLEAR-R* functions. *CLEAR-I* puts you in the insert mode and *CLEAR-R* puts you in replace mode. The mode is always displayed on the status line.

2 Files

DEFT Edit allows you to load text from ASCII files on tape or disk, edit the text and then write back to cassette or disk. In addition, you can use the write function to write to the printer.

2.1 Getting A File

The *CLEAR-G* (Get) function allows you to insert the contents of a file into the current text in front of the character that the cursor is currently positioned over. This allows you to both initially load a file and to merge several files in memory.

When you enter *CLEAR-G* you are prompted for a file name on the status line. When typing on the status line the only thing that you can do is enter characters, the left-arrow to backspace and the ENTER key to terminate the entry. The default suffix used by the editor is blanks. If you enter no file name then the function is aborted and you return to the editing session.

2.2 Writing A File

The *CLEAR-W* function is used to write the in-memory text to a file. Like the *CLEAR-G* you are prompted for a file name. However, you are given the default of the file name used in the last *CLEAR-G* operation. If you enter any key other than ENTER then the default entry is erased and the character you entered is processed. Like the *CLEAR-G* function, if you enter a null file name the function is aborted.

2.3 Quitting and Reentering

The *CLEAR-Q* function is used to quit the editor and return to BASIC. After entering the function, you should immediately get the OK prompt.

When leaving **DEFT Edit**, the contents of the text area are not changed (unless you forgot to protect memory from BASIC with BASIC's CLEAR statement). You can reenter the editor and answer the *INITIALIZE? (Y)* question with either an *N* or *n* and return to the point in your edit session that you left. This is convenient when you wish to do a DIR to determine which files are on the disk before saving off the text in memory.

2.4 Exiting

The *CLEAR-X* function allows you to combine the *CLEAR-W* and *CLEAR-Q* functions with a single function. The write function is performed followed by the quit function. The text in memory is left un-changed.

2.5 File Errors

When reading from or writing to a file, a number of errors can occur. Whenever an I/O error occurs the message *FILE ERROR...* is displayed on the status line and the editor waits for you to acknowledge seeing the message by depressing any key on the keyboard. This means that the first key depressed after the display of an error message will yield nothing more than the re-establishment of a normal status line presentation. Normal operation is then resumed. The possible error numbers are as follows:

- *-1, End of File* - You should not get this error number since an end of file is an expected occurence for **DEFT Edit**.
- *-2, I/O Error* - This indicates that some hardware oriented problem occured.
- *-3, File Not Found* - The file specified was not found.
- *-4, Illegal Operation* - This may occur if you try to read from the printer.
- *-5, Device Full* - There is no more space available on the specified device.

3 Pattern Processing

DEFT Edit contains commands for finding and changing text patterns.

3.1 Finding a Text Pattern

The *CLEAR-F* function is used to find a specific pattern in the text. After entering the *CLEAR-F* you are prompted on the status line for the string, of up to 24 characters, that you want to find. When typing on the status line the only things that you can type are characters, the left-arrow to backspace, and the *ENTER* key to terminate the entry.

When you depress the *ENTER* key the search will begin at the point in the text where the cursor was when you entered the *CLEAR-F* and will continue down to the end of the text. If a matching string is found then the line containing the string will be positioned at the top of the screen and the cursor will be positioned on the next character following the matching characters.

If you invoke the *CLEAR-F* function again, you will see that the prompt for the desired string defaults to the string that you entered on the last *CLEAR-F* or *CLEAR-C* function. You can just depress the *ENTER* key to find the next instance of the string in the text. If you type anything other than the *ENTER* key the old string will erase and you will be able to enter a new string.

If you enter no characters at all, no search will be made. If a search is made and the string is not found, the cursor will return to the point at which you entered the *CLEAR-F*.

3.2 Changing Text Patterns

In addition to finding a specific character pattern, you can change 1 or more occurances of one pattern to a second pattern. You use the *CLEAR-C* function to invoke this capability.

After entering the *CLEAR-C* you are prompted for the string to be searched for. After entering the string to be searched for, you are then prompted for the string that the first string is to be changed to. This can be 0 to 24 characters long. Finally, you are prompted for the number of occurances that are to be changed. If you don't enter any number, then the editor defaults to 1 occurance.

As each occurance is found and changed, it is displayed on the screen. When no more of the first string can be found, the function stops at the point where the last change was made. As in the

CLEAR-F function, if no first strings are found, the cursor will return to the point where it was when you entered the *CLEAR-C*. If you don't enter a first string, no changes are made.

4 Copying and Moving Text

There are 3 functions and a separate editor mode used to copy and/or move portions of text.

4.1 Marking and Saving Text

Before a portion of text can be copied and/or moved it must first be marked off and saved. This is done by positioning the cursor at either the first character or on the character following the last character of the text area to be saved. You then use the *CLEAR-M* function to mark that end of the area.

When you mark one end of a text area, two things happen. First, the mode changes to MARK to indicate that you are now marking an area of text rather than entering it. Second, the character that you marked is changed to a solid white square on the screen. This character will remain marked until you mark the other end of the text area.

Once you are in the mark mode, you cannot enter text. However, you can position the cursor with the arrows, CLEAR-Arrows, CLEAR-B, CLEAR-E and CLEAR-F functions. Once you have positioned the cursor to the other end of the text, you can mark it with the *CLEAR-M* function. The text that is saved starts with the mark that is closest to the beginning of the text and includes all characters down to but not including the mark closest to the end of text.

The mark function allows you to save up to 1.5K bytes of text in a separate in-memory *mark* buffer. If the marked area of text is greater than 1.5K bytes, then **DEFT Edit** prompts you for a name to give the file which it will create to save the marked text. This file name prompt occurs, provided the marked area is greater than 1.5K, immediately after the entry of the last *CLEAR-M* function. If a blank file name is entered then no action is taken and normal editing may be resumed.

4.2 Appending The Saved Text

Of course just saving the text away in a separate *mark* buffer or file doesn't do you much good unless you can do something with it. The *CLEAR-A* function allows you to append the text in the *mark* buffer into the screen text beginning in front of the current cursor position. The *CLEAR-A* function is not used to append text saved in a file, the *CLEAR-G* function is used instead when the text was saved in a file.

The contents of the *mark* buffer remain unchanged after this operation.

A typical copy operation would involve marking off the area of text to be copied, and then positioning the cursor to the point that it was to be copied to and invoking the *CLEAR-A* function.

If a section of text larger than 1.5K bytes needs to be copied into another area of a document, then the *CLEAR-M* function would be used to mark the text for copy. This would then yield a file name prompt for the file into which the saved text would be stored. Once the marked text is saved away, then the user would positon the cursor at the point in the document where the saved text was to be copied. The saved text would then be brought in with a *CLEAR-G* function followed by the name of the file containing the saved text.

4.3 Additional Mark Functions

When marking off a text area you can terminate the mark operation in 3 additional ways:

1. *CLEAR-D* may be used to mark the end of a text area. When used in this manner *CLEAR-D* is exactly like the *CLEAR-M* except that after saving away the text in either the mark buffer or a file, the area marked is deleted from the text. This provides the first half of a move operation rather than a copy. It can also be used to just delete areas of text.

2. *CLEAR-Q* terminates the mark operation without saving away any text. When the *CLEAR-Q* function is entered while the editor is in the mark mode, the mark operation is terminated with no action taken. The previous contents of the mark buffer are retained.

3. *CLEAR-W* allows you to save areas of text on a separate file or to print them on the printer. In this case the *CLEAR-W* function is entered to mark the ending point of a text area. After entering *CLEAR-W* you are prompted for a file name to which the marked off text is to be written. The contents of the mark buffer are not affected. This function allows the user to save any size of text to be filed, whereas the normal mark operation will only put text into a file if the text area being saved is larger than 1.5K bytes.

Edit

DEFT Pascal Compiler

Compile

1 Introduction

The **DEFT Pascal** Compiler is a program that allows you to create machine language programs from Pascal language source programs created with **DEFT Edit** or your own ASCII file text editor. The **DEFT Pascal** Compiler's features include:

- Generation of machine language programs, directly executable by the 6809 micro-processor, from Pascal language statements and declarations. Compiled programs can run many times faster than interpretive BASIC programs.
- Practically all of standard Pascal's language elements are supported.
- Program source files may be read from either cassette or disk with the resulting object files written to either cassette, disk, or the printer.
- Powerful compiler directives which provide the user with valuable compilation and source listing options, such as the option of having the assembler language representations of Pascal statements printed between the Pascal statements on the compiled program's source listing.
- Fully recursive compilation, which yields such flexibility as no fixed limitations on the number of dimensions to an array or table.
- Supports generation of recursive applications; programs that contain procedures that call themselves.

2 DEFT Pascal Compiler Operation

The command *LOADM "PASCAL":EXEC* will load the **DEFT Pascal** Compiler into memory from disk drive 0 and begin its execution, which is in two phases. In the first phase you will see the **DEFT Pascal** Compiler's screen with all of its prompts. This phase prompts the user to enter information required by the compiler for program compilation.

Upon the entry of the last prompted field, **DEFT Pascal** begins its second phase of operation. In this phase **DEFT Pascal** reads the source module file, parses the program statements, generates the corresponding machine instructions, saves the machine program version in an object module file, and generates the program source listing. After completing this phase **DEFT Pascal** has finished its execution which is marked by the return of the BASIC OK prompt.

Each **DEFT Pascal** Compiler prompt and its possible replies are described in the following sections.

2.1 SOURCE:

SOURCE requires the entry of the name of the source file which contains the Pascal language program that is to be compiled. The default file name extension is PAS. This means that if there is no extension specified with the entered file name, then the compiler adds the default extension of PAS to the file name before searching for that file.

2.2 OBJECT:

OBJECT requires the name of the object file that is to be created by the **DEFT Pascal** Compiler to hold the newly created program object module. This can be either on tape or disk or the name can be ommitted entirely if you do not wish to create an object file. The default extension is OBJ. If you do not specify an extension with the file name entered here, then the **DEFT Pascal** Compiler will add the default "OBJ" extension to your file name prior to actually creating that file.

2.3 LIST:

LIST requires the name of the source listing file which is to be created by **DEFT Pascal** in its second phase of operation. This can be tape, disk, screen or printer or the name can be ommitted entirely if you do not wish to create a list file. In this case only source lines

with errors and the corresponding error messages will be output to the screen.

The default extension is LST. If you do not specify an extension with the file name entered here, then **DEFT Pascal** will add the default LST extension to your file name prior to actually creating that file.

2.4 DEBUG?:

DEBUG? asks you whether you wish to have debug information included in the resulting object file. If you intend to use **DEFT Debugger** to debug this program, then a *Y* response should be entered. A yes response to this question results in **DEFT Pascal** adding the debugger symbolic linkages to your program, therefore making the resulting object module larger than it otherwise would have been. If you don't want the debug information included, you can answer this prompt with either an "N", or "n". Anything other than "N" or "n" (for No) is taken to be "Y" (for Yes).

The debug information will make your program significantly bigger but will allow you to symbolically debug your resulting program if you answer the **DEFT Linker**'s *debugger?* question yes. If you specify yes to the **DEFT Pascal** compiler's *debug?* question and No to the **DEFT Linker**'s *debugger?* question, then the debug information will still be in the final binary image even though the debugger module is not present.

2.5 DIRECTIVE:

DIRECTIVE requires any **DEFT Pascal** Compiler directive that you would like to include before any source lines are read. The following section *Compiler Controls* describes all the possible compiler controls that you could enter here.

2.6 Compiler Execution

After you answer the *DIRECTIVE* prompt, the program will begin executing. The compiler requires that the file PASCALIB/EXT be present on disk drive 0 when the *SOURCE:* prompt is answered. When the compiler is finished executing, control will return to BASIC and you will get the OK prompt.

3 Source Listing

The following is a brief description of the **DEFT Pascal** Compiler's source listing.

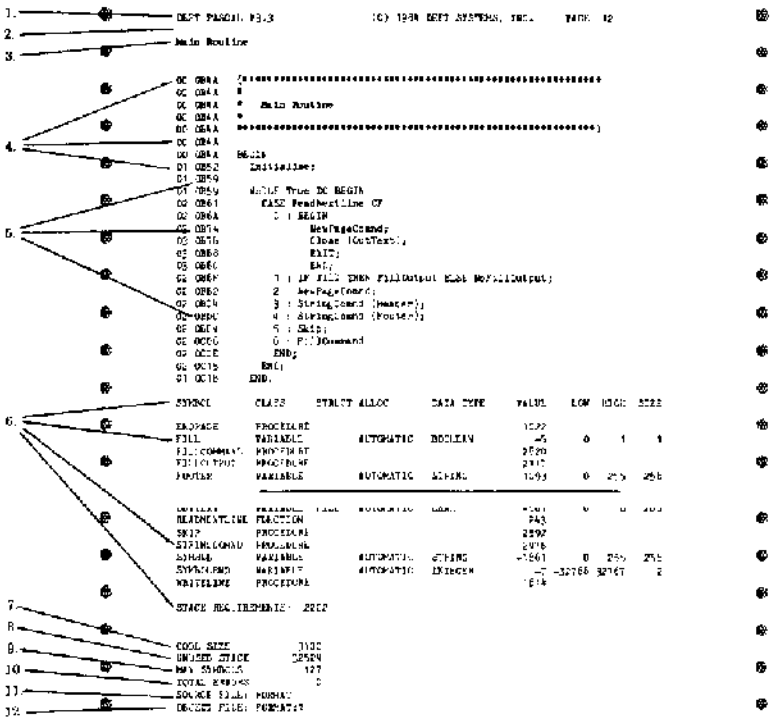
1. *Header* - This is the first line at the top of the source listing followed by the page number for that page of the listing
2. *Title* - This is the second line from the top of the source listing. The contents of this line are dictated by the programmer with a *title* directive.
3. *Subtitle* - This is the third line from the top of the source listing. The contents of this line are dictated by the programmer with a *subtitle* directive.
4. *Nesting Levels* - The first column of numbers printed with each line is actually two separate nesting levels:
   - The first one is the *procedure* nesting level. This identifies what level of *procedure* the current line of code is known in.
   - The second number is the *begin* nesting level. This identifies how many *begins* have been encountered so far with no matching *ends*.
5. *Program Location Counter* - This second column is a hexadecimal representation of the program address at which that line's executable statement will begin. All other numbers printed on the listing are decimal.
6. *Symbol Table* - A list of all the symbols that were defined within a Pascal block is produced at the end of each block. This list contains a number of fields for each of these symbols. Following are all the column headings and a description of the information printed under each heading:
   - *SYMBOL* - This is the symbol name.
   - *CLASS* - This identifies what kind of Pascal language element this symbol represents.
   - *STRUCT* - For structured types and variables, this column identifies what their structure is *(array, record, set, pointer* or *file).*
   - *ALLOC* - For variables, this column represents the allocation of that variable. Any *external* procedures or functions will have EXTERNAL printed here. Symbols which are fields

within a *record* will have the name of the corresponding *record* printed here.

- *DATA TYPE* - For variables, types and constants, The Pascal *type* specified for the data element represented by this symbol is printed under this heading.
- *VALUE* - This identifies the value of the symbol. For *static* variables, procedures, functions, labels and string constants it is the relative offset from the beginning of the module. For automatic variables it is the offset within the stack frame. For non-string constants, it is the value of the constant.
- *LOW* - This heading identifies the lowest or smallest value to which the data in a type or variable may be set. For arrays, it is the lowest possible subscript.
- *HIGH* - This heading identifies the highest or largest value to which the data in a type or variable may be set. For arrays, it is the highest possible subscript.
- *SIZE* - For variables, types and constants, this is the number of bytes of memory represented by the Pascal *type*.
- *STACK REQUIREMENTS:* - This title precedes the estimation of the number of bytes of stack space required to activate this block.

7. *CODE SIZE* - This is the fifth from the last line printed on the source listing. Following is the number of bytes of memory that the program will require when it is loaded.
8. *UNUSED STACK* - The following number is the amount of stack space available but unused by the compiler itself. As you create more symbols and deeper levels of nesting in your program, this number will grow smaller. This stack space essentially represents the limits of the compiler for number of symbols and levels of nesting (of all kinds).
9. *MAX SYMBOLS* - The following number is the maximum number of symbols known at any point in the Pascal source program. Due to pre-defined symbols and the definitions in PascalIB/EXT, there will always be over 60 symbols defined in a program. Note that each symbol definition takes up about 30 bytes of compiler stack space.

10. *TOTAL ERRORS* - This is the number of compilation errors.
11. *SOURCE FILE* - Following this is the name of the source file containing the program source statements which generated this listing.
12. *OBJECT FILE* - Following this is the name of the file which contained the program object at the end of this compilation.




```
DEMO PASCAL V3.3                (C) 1988 DEMO SYSTEMS, INC.     PAGE  12

Main Routine

0C 0B4A  {************************************************************
0C 0B4A  *
0C 0B4A  *  Main Routine
0C 0B4A  *
0C 0B4A  ************************************************************}
0C 0B4A
00 0B4A  BEGIN
01 0B52    Initialize;
01 0B59
01 0B59    WHILE True DO BEGIN
02 0B61      CASE ReadNextLine OF
02 0B6A        0 : BEGIN
03 0B74              NewPageCmnd;
03 0B7E              Clear (OutText);
03 0B88              EXIT;
05 0B90            END;
06 0B97        1 : IF FILE ENDS FillOutput ELSE WriteLN(output);
07 0B9E        2 : NewPage(comm);
07 0BA4        3 : StringCommd (header^);
07 0BAC        4 : StringCommd (footer^);
0F 0BC4        5 : Skip;
0E 0BCC        6 : FillCommand
03 0BDE      END;
06 0BE5    END.

SYMBOL      CLASS     STRUCT ALLOC      DATA TYPE      VALUE    LOW  HIGH  SIZE

EXCPAGE     PROCEDURE                                   1022
FILL        VARIABLE          AUTOMATIC  BOOLEAN          -5    0     1     1
FILLCOMMAND PROCEDURE                                   2020
FILLOUTPUT  PROCEDURE                                   2012
FOOTER      VARIABLE          AUTOMATIC  STRING         1093    0   255   256

INITIALIZE  PROCEDURE                                   1E3E
READNEXTLINE FUNCTION                                    943
SKIP        PROCEDURE                                   2092
STRINGCOMMD PROCEDURE                                   2078
SYSNULL     VARIABLE          AUTOMATIC  STRING        -1661    0   255   255
SYSVALCMD   VARIABLE          AUTOMATIC  INTEGER          -1 -32768 32767     2
WRITELINE   PROCEDURE                                   1E1A

STACK REQUIREMENTS:  2212

CODE SIZE          3102
UNUSED STACK      32524
DATA SEGMENTS       127
TOTAL ERRORS          0
SOURCE FILE: PASMAIN
OBJECT FILE: PASMAIN
```

4 Compiler Controls

Compiler controls are those instructions included in your source code or in the *DIRECTIVE:* prompt which direct the compiler's operation rather than the resulting program's operation. A compiler control is a source line with a percent sign (%) as the first character in the line. The control itself is a single character following the %. Any required parameters then follow the control character. For those controls not requiring parameters, additional controls may be included in consecutive columns. The % is not required in the DIRECTIVE prompt.

4.1 Listing Control

DEFT Pascal normally produces a source line listing file. The List (L) and Nolist (N) compiler controls allow you to control which portions of the source lines are included in this listing. These controls are additive; that is, if you include more than one list or nolist control in a row, it takes an equivalent number of the other to cancel its effects.

This additive nature gives you the ability to pre-cancel an imbedded nolist command with a preceding list command and vice-versa. This is very convenient when using copy files (see below). For example, **DEFT Pascal** copies by default the file PASCALIB/EXT which has a nolist control at the beginning of the file and a list command at the end. You can "unsuppress" its listing by including a list (L) control in response to the DIRECTIVE prompt in the compiler start-up screen.

4.2 Assembler Listing Control

DEFT Pascal is a true Pascal source to 6809 object code compiler. As such, it can produce a listing of the corresponding assembly language code that would be required to produce the same object. The default condition for the compiler is to not produce this assembly language listing. The compiler control used to turn on this listing is the plus sign (+). The compiler control used to turn it off is the minus sign (-).

4.3 Top of Page

The source listing produced by the **DEFT Pascal** Compiler normally prints 55 lines per page. However, you can force the compiler to start a new page at any point by including the eject (E) compiler control.

4.4 Title and Subtitle

Included at the top of each page produced by the compiler is the compiler's name, copyright notice and page number. In addition, on the following two lines you can specify a title (T) and subtitle (S). The remainder of the line on which the control is specified becomes the title or subtitle. Following are examples:

```
%T This is a Title String
%S This is a Subtitle String
```

Note that the presence of either control implies an eject (E). Blanks immediately following the control up to the first non-blank are suppressed in the actual title or subtitle.

In addition to printing at the top of each page, the title string is also included as a comment statement in the resulting object file. It will then also appear in **DEFT Linker's** listing file.

4.5 Copy

Sometimes it is desirable not to include your entire program in a single source file even though you wish to compile it as a single unit. This may be due to limitations of the editor or to allow common definitions for *interface* modules (see *Separate Compilation).*

The copy (C) compiler control allows you to tell the compiler where additional source lines should be taken from. The remainder of the control line is considered to be the file name of a Pascal source file. The compiler will read all the lines in the specified source file before reading the next line in the current source file. Example:

```
%C GRAPHINT:1
```

This line causes the file GRAPHINT/PAS on disk drive 1 to be completely read before reading the next line in the current file. Note that copy controls can be nested. That is, a file that is copied may itself contain a copy control. This nesting is only allowed to two levels.

Compile

DEFT Macro/6809 Assembler

Asm

1 Introduction

The **DEFT Macro/6809** Assembler is a program that allows you to create machine language programs from Motorola 6809 Assembler language source programs created with **DEFT Edit. DEFT Macro/6809's** features include:

- Generation of machine language programs, directly executable by the 6809 micro-processor from Motorola 6809 Assembler language statements. Assembled programs can run up to 1000 times faster than interpretive BASIC programs.
- Separate assembly facilities which enable you to break up a large program and assemble it in pieces. These pieces can be written in either **DEFT Macro/6809** assembly language or **DEFT Pascal.**
- Assembler directives which provide the user with valuable assembly and source listing options.
- Powerful macro facilities which allow the user to define inline code sequences with one *macro instruction* in the source program.

2 6809 Macro Assembler Operation

The command *LOADM "ASSEMBLE":EXEC* will load **DEFT Macro/6809** into memory from disk drive 0 and begin its execution, which is in two phases. In the first phase you will see the **DEFT Macro/6809** screen with all of its prompts. This phase prompts the user to enter information required by the assembler for program assembly.

Upon the entry of the last prompted field, **DEFT Macro/6809** begins its second phase of operation. In this phase it assembles the source language program statements into a machine program in two passes. In the first pass, **DEFT Macro/6809** reads the source module file and generates the symbol table. In the second pass, it generates the corresponding machine instructions, saves the machine program version in an object module file, and generates the program source listing. After completing this pass, **DEFT Macro/6809** has finished its execution which is marked by the return of the BASIC OK prompt.

Asm

Each **DEFT Macro/6809** prompt and its possible replies are described in the following sections.

2.1 TITLE:

TITLE: requires the string of characters that you want to see at the top of each page of your assembly listing. You do not have to enter a title if you don't want to, but it does come in handy when you want to identify a source listing file at a glance.

2.2 SOURCE FILE:

SOURCE FILE: requires the entry of the name of the source file which contains the 6809 assembler language program that is to be assembled. The default file name extension is ASM. This means that if there is no extension specified with the entered file name, then **DEFT Macro/6809** adds the default extension of ASM to the file name before searching for that file.

2.3 OBJECT FILE:

OBJECT FILE requires the name of the object file that is to be created by **DEFT Macro/6809** to hold the newly created program object module. This can be either on tape or disk or the name can be ommitted entirely if you do not wish to create an object file. The default extension is OBJ. If you do not specify an extension with the

file name entered here, then **DEFT Macro/6809** will add the default *OBJ* extension to your file name prior to actually creating that file.

2.4 LIST FILE:

LIST FILE: requires the name of the source listing file which is to be created by **DEFT Macro/6809** in its second phase of operation. This can be tape, disk, screen or printer or the name can be ommitted entirely if you do not wish to create a list file. The default extension is LST. If you do not specify an extension with the file name entered here, then **DEFT Macro/6809** Assembler will add the default *LST* extension to your file name prior to actually creating that file.

2.5 Assembler Execution

After you have answered the LIST FILE: prompt the assembler will begin its first pass. During this first pass only the disk will appear to be doing anything. The assembler will begin printing on its second pass through the source code.

3 Source Listing

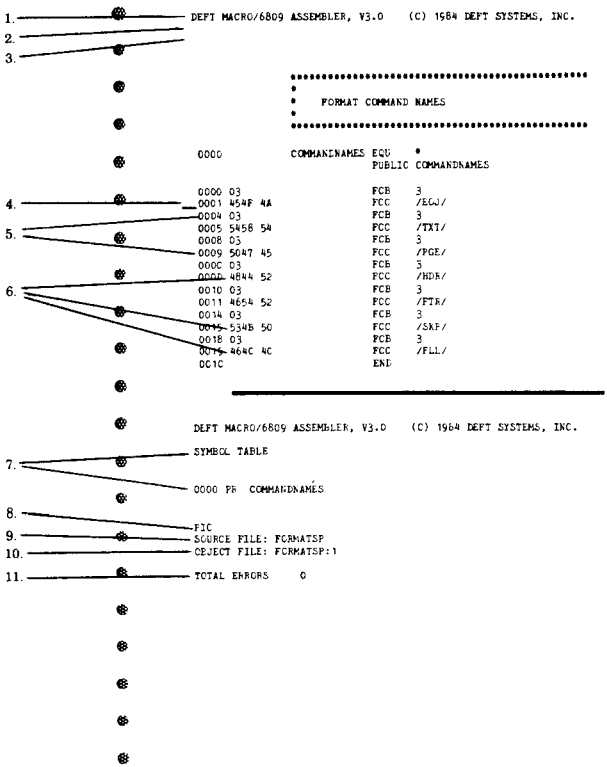
The following is a brief description of the **DEFT Macro/6809** Assembler's source listing.

1. *Header* - This is the first line at the top of the source listing followed by the page number for that page of the listing.
2. *Title* - The contents of this line are dictated by the programmer with a *title* directive.
3. *Subtitle* - The contents of this line are dictated by the programmer with a *subtitle* directive.
4. *Addressing Indicator* - This is an alphabetic character which prefixes the *Location Counter* to indicate how the instruction at that location is making a reference. An R indicates that an external relative reference is being made. An X indicates that an external absolute reference is being made. An N indicates that a local relative location is being referenced in an absolute mode.
5. *Location Counter* - This is the four digit number which immediately follows the line number. This four digit number is the hexadecimal representation of the program relative address at which this source code instruction would begin.
6. *Object Representation* - The set of numbers which immediately follows the location counter is a hexadecimal representation of the assembler instruction after the instruction has been converted into the object file machine language format. The very first two digits of this field represent the instruction's opcode. The remaining digits of this field represent the instruction's operands, where applicable.
7. *Symbol Table* - At the end of every assembler program, 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 relative address which the symbol is used to reference.
   - *Symbol Type* - This is the one to three character field which immediately follows the symbol value. This field identifies whether a symbol represents an absolute value (A), a program relative value (R), an external address (X), a public address

(P), 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.

8. *Position Independence* - This is the third from the last line printed on the source listing. The character expression found on this line identifies whether the assembled program is position independent or non-position independent. PIC indicates that the resulting machine program contained in the program's object file is *Position Independent Code.*
9. *SOURCE FILE* - This is the name of the source file containing the program source statements which generated this listing.
10. *OBJECT FILE* - This is the name of the file which contained the program object at the end of this assembly.
11. *Total Errors* - This is the last line printed on the program source listing and is the decimal number of errors encountered by **DEFT Macro/6809** during program asssembly.



```
1.  DEFT MACRO/6809 ASSEMBLER, V3.0    (C) 1984 DEFT SYSTEMS, INC.       PAGE    1
2.
3.
                        ********************************************
                        *
                        *    FORMAT COMMAND NAMES
                        *
                        ********************************************

0000                    COMMANDNAMES EQU    *
                                     PUBLIC COMMANDNAMES

0000 03                              FCB    3
4.  0001 454F 4A                     FCC    /EOJ/
0004 03                              FCB    3
5.  0005 5458 54                     FCC    /TXT/
0008 03                              FCB    3
0009 5047 45                         FCC    /PGE/
000C 03                              FCB    3
000D 4844 52                         FCC    /HDR/
6.  0010 03                          FCB    3
0011 4654 52                         FCC    /FTR/
0014 03                              FCB    3
0015 534B 50                         FCC    /SKP/
0018 03                              FCB    3
0019 464C 4C                         FCC    /FLL/
001C                                 END

DEFT MACRO/6809 ASSEMBLER, V3.0    (C) 1984 DEFT SYSTEMS, INC.       PAGE    2
7.  SYMBOL TABLE

    0000 PR  COMMANDNAMES

8.  FIC
9.  SOURCE FILE: FORMATSP
10. OBJECT FILE: FORMATSP:1
11. TOTAL ERRORS    0
```

DEFT Linker

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.

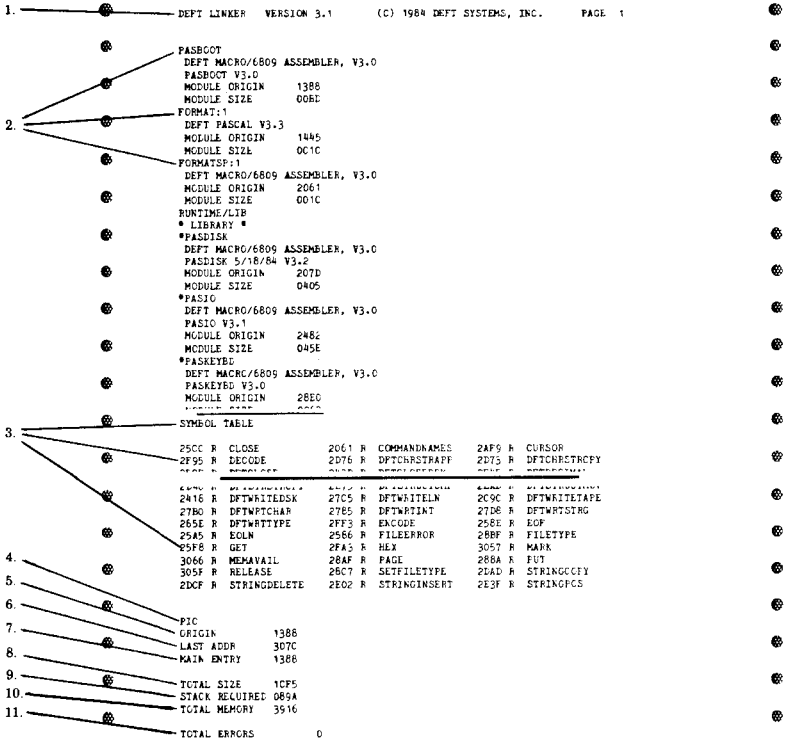
7. *MAIN ENTRY* - The four digit number following this title is the hexadecimal representation of the first address in memory where this program begins its execution.

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.

DEFT Debugger

Debug

1 Introduction

The **DEFT Debugger** is a software module which can be linked into any program produced by **DEFT** software products. It becomes the *main* module in the resulting program and allows the programmer to control its resulting execution. **DEFT Debugger** includes the following features:

- Like other debuggers, this one provides for memory and register display and modification as well as instruction breakpoints. Memory display and modification can occur in hex, decimal, floating point, ASCII and string formats.
- Single Pascal statement execution is available when the DEBUG option is specified at compile time.
- Normal program operation can be interrupted and the Debugger activated when the *BREAK* key is depressed.
- Symbolic access to memory areas is automatically provided by a special interface to the **DEFT Pascal** Compiler. This symbolic access includes automatic as well as static variables.
- A general expression capability allows the Debugger to perform all arithmetic and type and base conversions for you.
- A trace facility provides you with a procedure call history so that you can see how you got to a specific point in a Pascal program.
- Automatic screen preservation restores the screen area and attributes anytime program execution is resumed. This simplifies debugging of graphic programs.

2 General Operation

Although there are a number of features built into the **DEFT Debugger** specifically to debug Pascal programs, any program produced with DEFT software products can be debugged with it.

2.1 Linking in DEFT Debugger

In order to use **DEFT Debugger** you answer the **DEFT Linker's** *DEBUG? (Y)* question with anything other than *N* or *n* when you link the program. **DEFT Debugger** is automatically included in the resulting binary and gets initial control of the 6809 microprocessor when your program is executed. **DEFT Linker** provides **DEFT Debugger** with a table of all the module names and offsets in the resulting program along with the address where your program would normally begin execution. **DEFT Debugger** is loaded, as a part of your program, when you load your program with the *LOADM "myprogm":EXEC* command.

2.2 Debug Screen

After linking your program you are ready to execute it. When you begin execution **DEFT Debugger** will gain control and present you with its screen. This initial screen looks like this:

```
SYMBOLIC ONLINE DEBUGGER V3.x
(C) 1983 DEFT SYSTEMS, INC.
COMMAND:
 PS  02     B0 0000
 VD  00     B1 0000
 VC  00     B2 0000
            B3 0000
 CC  xx     B4 0000
 A   xx     B5 0000
 B   xx     B6 0000
 DP  xx     B7 0000
 X   xxxx
 Y   xxxx
 U   xxxx
 PC  xxxx
 S   xxxx
```

DEFT Debugger is now waiting for a command to execute and has displayed the complete set of registers it maintains for the program being debugged. You will normally enter a two character command. **DEFT Debugger** then prompts you for any additional

parameters required by the particular command.

The chapter on *Commands* describes all the commands and their required parameters. The chapter on *Expressions* describes the rules for forming expressions which are used in most parameters. What you see on the screen when the Debugger is first activated or anytime you hit a breakpoint is the automatic execution of the *DR* command. Following is a short description of the types of operations for which you might use **DEFT Debugger.**

2.3 Setting Breakpoints

One of the first things that you will want to do with the Debugger will be to set a breakpoint. A breakpoint is a place in your program where you want your program's execution to be suspended and **DEFT Debugger** activated. This allows you to examine variables or in the case of assembler language, registers. You can then see if the program has produced the proper intermediate results.

You set a breakpoint by using the Debugger's *modify register* command to set the value of one of the eight Breakpoint registers to the address of the place in your program where you want the breakpoint to occur. You have 8 breakpoint registers which allows you to specify up to 8 different places in your program at one time. This is especially convenient when you are not sure which place your program will go to first. The section on *Symbols* under *Expressions* describes how to specify a symbolic address.

2.4 Executing Your Program

After having set some (possibly no) breakpoints, you may then use **DEFT Debugger**'s *GO* command to begin (or continue) your program's execution. Another possible command is **DEFT Debugger**'s *ST* (Single Step) command which will allow you to specify the number of Pascal statements that you want to execute. Note that this option is only available when you have previously enabled the *debug?* option when the Pascal program was compiled.

2.5 Interrupting Program Execution

If you used the *GO* command to start execution, it will stop executing when one of the breakpoints that you specified is encountered. If you used the *ST* command, then execution will stop when the specified number of Pascal statements have been executed.

In either case you may stop the program's execution by depressing the *BREAK* key. If the program was compiled with the *DEBUG* option enabled then execution will stop on the next Pascal statement that is executed. Depressing the *BREAK* key while the program is prompting for keyboard input will cause it to stop even if the Debug option was not enabled at compile time.

2.6 Displaying/Modifying Memory and Registers

After your program stops, the Debugger is re-activated and you can use the display commands to determine what your program has done so far. You can change any variable or register that you wish before resuming execution again in order to change the way that your program is executing. Note that if your program stopped because it encountered a breakpoint that you specified via one of the breakpoint registers, then you will have to clear that breakpoint before resuming your program. Otherwise, the program will immediately breakpoint again.

2.7 Checking Program State

In addition to variables (memory) and registers, you can also use the *US* (User Screen) command to see what the screen is supposed to look like when **DEFT Debugger** is not using it. In addition, the *TR* (TRace) command will follow the chain of pointers that Pascal builds on the stack. This trace of all the current activation blocks will tell you what Pascal procedures are currently active and where they were called from.

3 Commands

This section describes all the commands available on **DEFT Debugger**. The title of each subsection names the corresponding command and contains the two character command representation in parentheses.

3.1 Display Register (DR)

This command causes all the **DEFT Debugger** registers to be displayed. All registers are displayed in hexadecimal. Those which are 16 bit registers are also displayed as module offsets with the module name and hex offset displayed following the absolute hex value.

The registers B0 through B7 are the breakpoint registers which can be set to addresses in your program at which you want execution to stop. The registers CC, A, B, DP, X, Y, U, PC and S are the 6809 machine registers. The remaining three registers relate to the graphic capabilities of the TRS-80 Color Computer and are as follows:

- *PS* is the Page Select register. The *lower* 7 bits of this register specify the *upper* 7 bits of the memory address at which the screen page begins. This value is initially 2 indicating that the screen page begins at address $400 or 1024.
- *VD* is the Video Display Generator register. The lower 3 bits of this register specify the graphics mode that is to be used.
- *VC* is the Video Control register. The *upper* 5 bits of this register specify the color set and qualify the graphics mode selected by the VDR.

Unlike the 6809 registers, the graphics registers cannot be read and saved by **DEFT Debugger**. Therefore anytime your program modifies these values at a point at which you are breakpointing, you will have to tell the Debugger what these values should be. This is done via the *Modify Register (MR)* command.

3.2 Display Word (DW)

This command allows you to display 1 or more 16 bit words in memory in both decimal and ASCII formats. There are two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the first 16 bit word to display.

- *COUNT:* - This parameter requires an expression which specifies the number of 16 bit words to display. If you enter nothing then the count defaults to 1.

3.3 Display Byte (DB)

This command allows you to display 1 or more 8 bit bytes in memory in both decimal and ASCII formats. There are two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the first 8 bit byte to display.
- *COUNT:* - This parameter requires an expression which specifies the number of 8 bit bytes to display. If you enter nothing then the count defaults to 1.

3.4 Display Floating Point (DF)

This command allows you to display a Pascal floating point (real type) number variable. There is one parameter:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the floating point variable.

The floating point variable is displayed in decimal format.

3.5 Display String (DS)

This command allows you to display a Pascal string variable. There is one parameter:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the string variable.

The string variable is displayed in ASCII format. In addition, the decimal length of the string is displayed.

3.6 Display Variable(DV)

This command allows you to display a variable as either a word, byte, floating point or string. You must use a symbol as part of the *ADDRESS* parameter. **DEFT Debugger** uses the type of the symbol used to determine which type of display to perform. There are two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the variable.
- *COUNT:* - This parameter is prompted for only when the symbol type is an ARRAY. It requires an expression which specifies the number of 8 bit bytes or 16 bit words to display. If you enter nothing then the count defaults to 1.

3.7 Display Hex (DH)

This command allows you to display 80 bytes of memory in both hex and ASCII representation. There is one parameter:

- *ADDRESS:* - This parameter requires an expression which specifies the address of memory to begin the display.

This command displays the memory as 10 lines of 8 bytes each. The last 3 hex digits of the memory address is displayed at the beginning of each line followed by the hex representation of the 8 memory bytes at that location. Finally, the ASCII representation of those same bytes is displayed at the end of the line.

3.8 Display Next (DN)

This command is almost exactly the same as Display Hex (DH) except that you are not prompted for an address. The display begins at the point where the last Display Hex or Display Next left off. This command provides a convenient means to page through memory.

3.9 Modify Register (MR)

This command allows you to modify any of **DEFT Debugger**'s registers. All registers displayed on the Display Register screen can be modified. This command has two parameters:

- *REGISTER:* - This parameter requires the 1 or 2 character name of the register that is to be modified.
- *VALUE:* - This parameter requires an expression which is the value that the register is to be set to.

3.10 Modify Word (MW)

This command allows you to modify a 16 bit word in memory. It requires two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the 16 bit word to modify.
- *WORD xxxx VALUE:* - This prompt shows the hexadecimal address that will be modified (the "xxxx"). It requires an expression which specifies the value that the word at that location is to be set to. If nothing is entered, the command is terminated and the word is not modified. If a value is entered, then the word is modified and **DEFT Debugger** continues to prompt for subsequent words until nothing is entered.

3.11 Modify Byte (MB)

This command allows you to modify an 8 bit byte in memory. It requires two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the 8 bit byte to modify.
- *BYTE xxxx VALUE:* - This prompt shows the hexadecimal address that will be modified (the "xxxx"). It requires an expression which specifies the value that the byte at that location is to be set to. If nothing is entered, the command is terminated and the byte is not modified. If a value is entered, then the byte is modified and **DEFT Debugger** continues to prompt for subsequent bytes until nothing is entered.

3.12 Modify Floating Point (MF)

This command allows you to modify a Pascal floating point (real type) number variable in memory. It requires two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the floating point number to modify.
- *VALUE:* - This parameter requires the decimal representation of the floating point value that is to be inserted.

3.13 Modify String (MS)

This command allows you to modify a Pascal string in memory. It requires two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the string to modify.
- *xxxx STRING:* - This parameter requires a number of ASCII characters to be entered. These are stored directly in the string with the number of characters entered becoming the string's length. If nothing is entered, the command is terminated and the string is not modified.

3.14 Modify Variable (MV)

This command allows you to modify a Pascal variable by identifying it symbolically. This command allows **DEFT Debugger** to determine whether to execute a *Modify Word, Modify Byte, Modify Floating* or *Modify String* command depending on the type of the variable named in the *ADDRESSS:* parameter.

3.15 Clear Breakpoints (CB)

This command is used to clear all the breakpoint registers to zero. You can set a breakpoint by using the Modify Register (MR) command to set one or more of these registers to a non-zero value. You can also clear an individual breakpoint by using the same command to set a breakpoint register to zero.

3.16 User Screen (US)

This command allows you to view the screen currently being displayed by the program under test. The values of the PS, VD and VC registers are used to determine what the display is to look like. The display persists until you type any character.

3.17 Evaluate (EV)

This command allows you to evaluate an expression and display its results in decimal, hexadecimal and ASCII. It requires one parameter:

- *VALUE:* - This parameter requires an expression which is to be evaluated.

3.18 Trace (TR)

This command allows you to see all the procedures which are currently active. The absolute address and module offset of the current program counter (PC) and each return address on the stack (beginning with the most recent) is displayed on each line. For those modules which also have symbols, the name of the procedure or function to which the return address points is also displayed. This then provides you with a list of each active Procedure/Function and the point in the calling Procedure/Function from which they were called.

Since this command relies on the standard Pascal frame structure, there are some limitations on its use:

- Only those Procedure/Function activations that have been completed will be displayed. If you set a breakpoint at the address of a Procedure or Function and then do a *TR*, you will not see that Procedure or Function in the list. You must set the breakpoint at (or Single Step to) the first statement in the Procedure or Function. Note that the Single Step *(ST)* command will not breakpoint in the middle of a Procedure/Function activation (unless you have set an explicit breakpoint).
- The command is not meaningful until after the complete activation of the main Pascal program. This is done the same as a Procedure or Function described above.
- Only the most recent 12 (or fewer) activations are listed.
- Calls to Assembly language routines will be listed only if they construct a Pascal frame structure on the stack.

3.19 Go (GO)

This command allows you to execute your program. If any of the breakpoint registers are non-zero then breakpoints are set at those points before program execution begins. **DEFT Debugger** will not regain control until one of the specified breakpoints is encountered. If one of the breakpoints is the same as the PC register then control will return immediately to the Debugger. This command has no parameters.

Once a breakpoint is encountered, the DR command is automatically executed and you are prompted for another

command.

3.20 Step (ST)

This command is similar to the *GO* command except that it uses the breakpoints inserted into the program by Pascal when you specified *debug* at compile time. Not only does the **DEFT Pascal** compiler include symbol tables, but it also generates a breakpoint instruction at the beginning of every Pascal statement when you specify the *debug* option. The *Step* command then lets you step through the Pascal statements by counting the corresponding breakpoints in the resulting code.

Note that this command will operate the same as the GO command if there are no Pascal modules with the *debug* option enabled. This command has 1 parameter:

- *COUNT*: - This parameter requires an expression which is the number of Pascal statements to execute before returning control to **DEFT Debugger**. If no expression is entered, a value of 1 is assumed.

3.21 Quit (QU)

This command allows you to terminate your program and return control to BASIC.

4 Expressions

Most DEFT **Debugger** commands will prompt you for some additional information such as an address of a field or a value which is to be used by the command. Most of these additional prompts require a general expression to be entered. This expression can be as simple as a single digit or as complex as several numbers in various bases with symbols combined with different operators. This section describes the rules for forming these expressions.

The **DEFT Debugger** deals entirely in 16 bit units. All components of an expression have 16 bit values and any resulting expression also has a full 16 bit value.

4.1 Constants

A constant used by itself is a legal expression. The **DEFT Debugger** supports 4 types of constants.

1. A *decimal* constant is a set of numbers in the range of -32768 to 32767.
2. A *hexadecimal* constant is a dollar sign ($) followed by up to 4 hexadecimal digits (0..9,A..F). If the constant is less than 4 digits long, leading zeroes are assumed.
3. An *ASCII* constant is a single quote (') followed by a single ASCII character. The value of this constant is the binary value of the ASCII character as the low 8 bits with the high 8 bits being zero.
4. A *double ASCII* constant is a double quote (") followed by two ASCII characters. The value of the constant is the binary value of the first ASCII character as the high 8 bits and the second as the low 8 bits.

4.2 Registers

The current contents of any of the registers can be referenced by entering a percent sign (%) followed by a one or two character register name. The available registers are those displayed via the Display Register (DR) command. They are as follows:

Mnemonic	Bit Size	Description
PS	8	Page Select
VD	8	Video Display Generator
VC	8	Video Control
CC	8	6809 Condition Code
A	8	6809 Accumulator A
B	8	6809 Accumulator B
DP	8	6809 Direct Page
X	16	6809 Index X
Y	16	6809 Index Y
U	16	6809 User Stack
PC	16	6809 Program Counter
S	16	6809 System Stack
B0	16	Breakpoint 0
.		
.		
.		
B7	16	Breakpoint 7

4.3 Symbols

Symbols are the names or *identifiers* that you used in your source code program to reference variables, procedures and functions. If the program that you are debugging has some Pascal modules in it, you can have the compiler include the symbols found in these modules by answering its *DEBUG?* prompt with anything other than *N* or *n*. This will cause the compiler to include the names of all the variables, procedures and functions in specially formatted tables. These tables are imbedded in the resulting object module code.

Object modules created with this option will be larger due to the presence of the symbols which will be part of the final load module binary code. When you have several Pascal modules in a single program, you can reduce the symbol table memory requirements by specifying debug symbols in only the modules that you wish to debug. The debugger knows which modules have symbols and which ones don't so that you only get the symbols that you need.

There are three types of symbols which are referenced in three different ways:

1. A *Module* symbol is the filename (not including the extension) of an object file or library section which was linked with the debugger. You indicate a module symbol with a leading less than sign (<) followed by the symbol itself. The names of all the object modules that are linked together are known to **DEFT Debugger** regardless of whether symbols internal to the corresponding module are present. This means that you can use module symbols even with assembly language modules. The value of a module symbol is the absolute memory address of the first instruction at the beginning of the module.

   One of the most common uses of a module symbol is to specify an address within a module. This is usually done as follows:

   <MYMODULE+$1A3

   This form can be used to set breakpoints in either Pascal or assembly language modules. In this case 01A3 is the offset within the module where the Pascal statement starts on which you want to breakpoint.

2. A module symbol can be further qualified with a *static symbol*. This is done by immediately following the module symbol with a greater than sign (>) followed by the static symbol. This static symbol can represent any Pascal procedure, function or statically allocated variable. The value of a static symbol is the beginning memory address of the program element represented by the symbol.

   A static symbol can be further qualified to any level required by entering additional greater than signs (>) followed by the qualifier. For example:

   <MYMODULE>UTILPROC>LCLFUNC>X

   This entry specifies the static symbol *X*, which is local to the function *LCLFUNC* which is contained within the procedure *UTILPROC*. This procedure in turn is in the module *MYMODULE*.

   After a module has been referenced (either by itself or as part of a static symbol reference) the next static symbol can be specified without specifying that same module name. **DEFT Debugger** will use the last module referenced, as a basis for its search, anytime a static symbol is specified without a leading module name.

3. An *automatic* symbol is indicated when a leading alphabetic character is detected. In this case **DEFT Debugger** will automatically scope the symbol by following the static procedure call links in the stack. This type of symbol specification will find the symbol which is known at the current point in the program. You can use this type of specification for procedure, function and static variable symbols as well as automatic variable symbols.

4.4 Terms and Indirection

The elements or arguments of an expression, *constants*, *registers* and *symbols*, are generically known as *terms*. You can add a level of *indirection* to a term by prefixing it with an at sign (@). This means that the *value* of the term is used identify the location of, or to *address*, a 16 bit word in memory. The contents of that memory word are then used as the value of the term. This is known as an *Indirect Term*.

4.5 Operators

Terms and Indirect Terms can be combined with the use of *operators*. The operators which are available are the four arithmetic operators: addition (+), subtraction (-), multiplication (*), and division (/). There is no precedence between operators and all expressions are evaluated from left to right.


Debug

DEFT Lib Object Librarian

1 Introduction

DEFT Lib is a program that creates and maintains libraries of object files. These object file libraries are then conditionally used by **DEFT Linker** when creating a binary load module file.

The purpose of linking with an object file library is to include only those portions of object code used by a given program. For example, if you have an object file created with one of the **DEFT** high level language compilers, then that particular program might not use strings or real arithmetic. When that object file is linked with the corresponding library, the object files for strings and real arithmetic will not be included in your final binary load module. However, object files for, say, I/O *would* be included.

DEFT Lib provides the following major features for library maintenance:

- Separate input and output libraries means that mistakes can be corrected by starting over.
- Object files can be added to a library in the form of library sections.
- **DEFT Lib** ensures that duplicately named sections are not added to the same library.
- Library sections can be deleted.
- Complete libraries can be merged together.
- A library can contain up to 50 sections.

2 Operation

Whenever you wish to create or update object libraries you can run **DEFT Lib**. The command *LOADM"LIB":EXEC* will load **DEFT Lib** from disk drive 0 and begin its execution. Once the program is loaded and the disk drive light has gone off, you may change diskettes if you wish.

DEFT Lib operates by reading in an old library file (if one exists) and copying it to a new library file. It is during the copy that the changes that you wish to make are actually performed. The old library file, is never modified by **DEFT Lib**. **DEFT Lib** operates in three phases.

During the first phase it prompts you for the old and new library files. It then prompts you for all the sections that you wish to delete from the old library as it is copied to the new library.

The second phase involves doing the actual copy and performing the requested deletes. It is during this phase that you will find out if any of the specified sections to be deleted were actually in the old library to begin with.

Once the copy is completed, the third phase will begin. **DEFT Lib** will prompt you for the names of the object files and object libraries that you wish to add to the new library. As you specify each name, **DEFT Lib** will make sure that it is not a duplicate and then add it to the new library, **DEFT Lib** will display each section name and ask if you want that section added to the new library. When duplicate section names are encountered, **DEFT Lib** will let you specify a new section name.

Following is a description of each prompt made by **DEFT Lib**.

2.1 OLD LIBRARY:

This is the name of an existing library file which is to be the primary source of information for creating the new library file. You do not have to enter an *old library* if you are creating a *new library* from scratch. The default file extension for this prompt is *LIB*.

2.2 NEW LIBRARY:

This is the name of the new library that **DEFT Lib** is going to create and which will contain the results of this update. You must enter a new library name and it must be different from the file name that you entered for *old library*. If you enter the same name as the *old*

library, you will destroy the old library file.

The default file extension for this prompt is *LIB*.

2.3 DELETE SECTION:

This is the name of a section in the old library that is not to be copied to the new library. You will only get this prompt if you specified an *old library* file name. After entering a section name (up to 8 characters), **DEFT Lib** will prompt you again for another section not to copy. **DEFT Lib** will let you enter up to 50 sections in this manner.

Once you have entered all the names that you wish not to appear in the *new library*, enter a null section name (just depress the *ENTER* key without entering any characters) to indicate that there are no more section names to be deleted.

2.4 ADD OBJECT FILE:

After the copying is completed, **DEFT Lib** prompts you for any object files that you would like to have added to the *new library*. At this point, **DEFT Lib** has finished using the *old library* file and you may remove the diskette containing it if you wish.

You may enter the name of either an object file, a library file or no file at all. If no filename is entered, **DEFT Lib** closes the new library and terminates execution. This is how you will tell **DEFT Lib** that you have no more object files or libraries to add. The default file extension is *OBJ*.

2.5 Adding an Object File

If you enter the name of an object file, **DEFT Lib** will open the file and then prompt you for the name of the section to use in the library. The prompt that you will get is:

SECTION NAME (nnnnnnnn):

The default section name is the name of the object file. This will be used if you do not enter a section name. You may use the *CLEAR* key to stop **DEFT Lib** from doing the add at this point if you wish. If the section name used (either the default or the one that you specified) is the same as one that is already in the new library, then you will receive the following prompt:

nnnnnnnn IS A DUPLICATE SECTION
NEW NAME:

You can enter a different name or you may use the *CLEAR* key to abort the add.

Once the section is added (or the add operation is aborted) you will get the *ADD OBJECT FILE:* prompt again.

2.6 Adding a Library File

If you enter the name of a library file, **DEFT Lib** will open the file and begin reading each section of the specified library. For each section found, **DEFT Lib** will then prompt you for the name of the section to use in the *new library*. The prompt that you will get is:

SECTION NAME (nnnnnnnn):

The default section name is the name of the section in the library file that you are adding from. This will be used if you do not enter a section name. You may use the *CLEAR* key to stop **DEFT Lib** from doing the add at this point if you wish. If the section name used (either the default or the one that you specified) is the same as one that is already in the new library, then you will receive the following prompt:

nnnnnnnn IS A DUPLICATE SECTION
NEW NAME:

You can enter a different name or you may use the *CLEAR* key to abort the add for this particular section.

After each section is added (or the add operation is aborted) you will get the *SECTION NAME:* prompt until all the sections have been read from the library that you are adding from. Once all the sections have been read, you will get the *ADD OBJECT FILE:* prompt again.

3 Error Messages

3.1 FILE IS NOT OBJECT OR LIBRARY

The file specified to an *ADD OBJECT FILE:* prompt was not a legal object or library file. The file is ignored.

3.2 I/O ERROR ON NEW LIBRARY

An I/O error occurred while **DEFT Lib** was writing to the new library file. If this occurs, **DEFT Lib** terminates execution. This error may be due to a bad diskette or because the diskette is full.

3.3 I/O ERROR ON OBJ/LIB FILE

An I/O error occurred while **DEFT Lib** was reading from the object or library file specified to the *ADD OBJECT FILE:* prompt. If an add was in progress, then it was only partially completed.

3.4 I/O ERROR ON OLD LIBRARY

An I/O error occurred while **DEFT Lib** was reading from the old library file. If this occurs, **DEFT Lib** terminates execution.

3.5 OPEN ERROR: n

An I/O error occurred while **DEFT Lib** was openning the specified old library, new library or object file. **DEFT Lib** will prompt for another file name. The *n* is an error number with one of the following values:

- *-1, End of File* - You should not get this error number since an end of file is an expected occurrence for **DEFT Lib.**
- *-2, I/O Error* - This indicates that some hardware oriented problem occurred.
- *-3, File Not Found* - The file specified was not found.
- *-4, Illegal Operation* - This may occur if you try to read from the printer.
- *-5, Device Full* - There is no more space available on the specified device.

DEFT Pascal Language

1 Introduction

The **DEFT Pascal** Compiler is a program which reads lines of source code produced with **DEFT Edit** (or any ASCII compatible editor) and produces a listing file and an object file. The object file produced contains actual machine codes which can be directly executed by the 6809 CPU in the CoCo after being linked by the **DEFT Linker.** This differs from a compiler which produces pseudo-code in the following respects:

1. The resulting program does not require an interpreter to execute. It is a self-sufficient program that requires only the Color Computer hardware.

2. The runtime execution environment is closer to assembler than BASIC. However, the **DEFT Debugger** provides some very powerful features which can make debugging the resulting machine language program almost as easy as debugging a BASIC program using the interpreter.

3. The performance of your program will be vastly better since each line of Pascal will result in only a few machine language instructions being executed. With an interpreter, several machine language subroutines within the interpreter will generally be executed per line of source code.

4. The program can be easily linked with **DEFT Macro/6809** assembly language modules and other **DEFT** high level language modules.

A Color Computer with 32K or 64K of RAM memory and 1 or 2 disk drives is a fairly powerful computer capable of most tasks being done on large micros and minicomputers. Using **DEFT Pascal** allows you to exploit that power to its fullest.

This section of the User's Guide describes those portions of **DEFT Pascal** which are ISO Standard. The following section, *Advanced Pascal*, describes the language extensions and assembler interface.

2 The Pascal Program

When programming in BASIC, there is almost no restriction on what order any of the statements must be placed. This is because almost all BASIC statements are *executable* statements. The only exception is the DIM statement, which is a *declaration* statement that defines arrays before they are used. DATA statements are neither *executable* nor *declaration* statements but they do represent a portion of the programs data. One of the primary aspects of the Pascal language is the presence of a very powerful declaration syntax, which requires that all Pascal programs be written in a specific format.

2.1 Block Structure

In Pascal, a program's structure is defined via a number of different types of *declaration* statements. These *declaration* statements allow a programmer to create an environment, or program structure, in which to get his job done with any number of the different types of *executable* statements. This provides the programmer with the ability to create a customized program structure that can match the problem structure of each program that he writes.

Pascal programs require the following elements in this order:

PROGRAM <program heading>;

<declaration statements>

BEGIN

<executable statements>

END.

Throughout this manual, words or phrases enclosed in <> are *non-terminators*. That is, they refer to a class of objects any one or more of which may be substituted at the place where the non-terminator is found. In the example above, *PROGRAM* is a *terminator* which represents exactly itself, whereas <program heading> is a non-terminator and represents some overall program information which will vary from program to program.

The important items in the structure are the <declaration statements> which define elements of your program and the <executable statements> which actually perform work on the defined elements.

Another way of describing the structure of a Pascal program is as follows:

PROGRAM <**program heading**> ; <**block**> .

Where <block> is equivalent to:

<**declaration statements**>

BEGIN

<**executable statements**>

END

This concept of *block* is central to the overall philosophy of Pascal. With this structure, <declaration statements> can define sub-blocks which in turn can themselves contain <declaration statements> which can further define sub-sub-blocks, and so forth and so on. It is with this hierarchy of *blocks* that the overall program is broken down into manageable pieces and implemented.

Block execution is initiated when that *block* is invoked or *activated*. Execution within a block starts with the first statement following the *begin* and proceeds sequentially with each of the following statements. (NOTE: in the section on *Compound and Control Statements* you will see how the order of execution can be altered). When the *end* statement is executed, the block is deactivated and control returns to the point at which the block was invoked.

Program execution starts at the last *begin* statement defined in the program. The program's execution will terminate at the last *end* statement defined in the program. Another way of putting it is that the last section of executable statements defined within a program is the first section to be executed. The sub-blocks, sub-sub-blocks, etc., which are the defined *procedures* and *functions* of the *program*, are *activated* by being invoked, or *called*, during the execution of the *program* or one of the previously executing *procedures* or *functions*.

2.2 Scope

The <declaration> statements within a *block* define <identifiers>, or data names, which are used by the <executable statements> within that *block*. When the *block* is activated, these <identifiers> are *activated* and become *known*. When the *block* is deactivated, the <identifiers> are *deactivated* and become *unknown*.

Identifiers used by <executable statements> may be either those defined by <declaration statements> within the same <block> or those defined in an enclosing <block>. All <identifiers> defined within all other

blocks of the program become *unknown.*

Note also, that the same identifier may be redefined in different levels of blocks. At any point in the program the innermost definition known at that point will be used. The following is an example.

```
PROGRAM Example;
VAR I,J : Integer;

   PROCEDURE Proc1;
   VAR I : Integer;

      PROCEDURE Proc2;
      VAR J : Integer;
      BEGIN      (* Proc2 BEGIN *)
        I := J
      END;

   BEGIN      (* Proc1 BEGIN *)
     Proc2;
     I := J
   END;

BEGIN      (* PROGRAM Example BEGIN *)
  Proc1;
  I := J
END.
```

The above statements are discussed in detail later in the manual, but for purposes of this example, short definitions are provided here. The *var* statements declare integer variables named *I* and/or *J*. The *I:=J* means that the value in *J* is assigned to *I*. The *Proc1;* and *Proc2;* statements invoke the corresponding procedures.

When the program first begins execution (just after the last *begin*) only the *I* and *J* and the procedure *Proc1* declared within the program *Example* are known. When *Proc1* is invoked and begins executing, *Proc2* becomes known, the *I* declared within *Proc1* becomes known, the *I* within the program *Example* becomes unknown (because of the temporary redefinition of *I*) and the *J* defined in the program *Example* remains known. When *Proc2* is invoked and begins executing, the *J* definition in *Proc2* temporarily replaces the *J* defined within program *Example.*

When *Proc2* returns and is deactivated, the previous *J* definition is restored. When *Proc1* is deactivated the previous *I* definition is restored and *Proc2* becomes unknown. Note that the above definitions and redefinitions apply to any type of <declaration statement> described below.

In *Advanced Pascal* you will find extensions to this fundamental structure.

2.3 Declaration Statements

As shown above, declaration statements come before the executable statements and are separated from them with the *begin* reserved word. The following are the <declaration statements>:

LABEL	<**identifier**>, ... ,<**identifier**>;
CONST	<**identifier**> = <**constant**>;
	.
	.
	.
TYPE	<**identifier**> = <**type definition**>;
	.
	.
	.
VAR	<**identifier**> : <**type definition**>;
	.
	.
	.
PROCEDURE	<**identifier**> <**parameter definition**> ;
	<**block**> ;
FUNCTION	<**identifier**> <**parameter definition**> :
	<**type definition**> ; <**block**> ;

As in most Pascals, the above declarations may occur in any order; although according to standard Pascal, the above order must be followed with the exception of the *PROCEDURE* and *FUNCTION* declarations, which can be mixed with each other. Note that the above definition is *recursive* in that <declaration statements> are part of both *procedures* and *functions*, both of which are themselves types of <declaration statements>.

More detailed information about each of the above declaration statements can be found in the chapter on each statement.

2.4 Executable Statements

Executable statements are placed after the *begin*. The first statement following the *begin* is the first statement actually executed by the resulting program. Following is a list of the executable statements:

<identifier> := <expression>

BEGIN <executable statements> END

CASE <expression> OF
<constant list> : <executable statement>;
...
<constant list> : <executable statement>
ELSE <executable statement>
END

FOR <identifier> := <expression> TO <expression> DO
<executable statement>

FOR <identifier> := <expression> DOWNTO <expression> DO
<executable statement>

GOTO <label>

IF <boolean expression> THEN <executable statement>
ELSE <executable statement>

READ (<file specifier> <input list>)

READLN (<file specifier> <input list>)

REPEAT <executable statements> UNTIL <boolean expression>

WHILE <boolean expression> DO <executable statement>

WITH <record variable> DO <executable statement>

WRITE (<file specifier> <output list>)

WRITELN (<file specifier> <output list>)

<procedure identifier> <parameter specification>

Anywhere that you see <executable statements> (plural) you can use the following:

<executable statement>;

.
.
.

<executable statement>

Note that the semicolon (;) is used to *separate* rather than *terminate* individual statements. Multiple statements separated by semicolons are allowed in both *begin* and *repeat* statements. The *else* clauses in both the *if* and *case* statements are optional and may be omitted.

Complete details on each of the above statements can be found in *Expressions and Assignments*, *Control*, *Procedures and Functions*, and *Input/Output*.

2.5 Program Statement

As shown above, the *program* statement is the first statement of your Pascal program. It has the following format:

PROGRAM <identifier> [(<identifier> , ... , <identifier>)] ;

The first <identifier> is the *program name* and serves no other purpose within the program. Following this is an optional parameter list enclosed in parentheses. In standard Pascal, this list identifies those file variables declared within the program which represent *external* files. The pre-defined file variables *input* and *output* must be present in this list if used (explicitly or implicitly) within the program.

In **DEFT Pascal**, the optional parameter list is allowed but ignored. This is because *all* files within a **DEFT Pascal** program are assumed to be external.

3 Language Elements

Before describing a Pascal program, it is necessary to describe the fundamental elements which make up one. Like BASIC, the Pascal language is constructed from the ASCII character set used on the Color Computer. These are as follows:

ABCDEFGHIJKLMNOPQRSTUVWXYZ	<upper case characters>
abcdefghijklmnopqrstuvwxyz	<lower case characters>
0123456789	<numbers>
!"#$% ' &'()*+,-./:;<=>?@[]	<special characters>

All the following definitions will be in terms of these characters. Note that except in character and string constants (defined below), there is no distinction between upper and lower case characters for those language elements using letters.

3.1 Reserved Words

Reserved words are groups of upper or lower case characters whose meaning has been predefined in the language. The following is a list of all the reserved words used in DEFT Pascal:

ABS	AND	ARRAY
BEGIN	BYTE*	CALL*
CASE	CHAR	CHR
CONST	DIV	DO
DOWNTO	ELSE	END
EXIT*	EXTERNAL*	FILE
FOR	FORWARD	FUNCTION
GOTO	IF	IN
INTERFACE*	LABEL	LSL*
LSR*	MOD	MODULE*
NEW	NOT	ODD
OF	OR	ORD
PACKED	PRED	PROCEDURE
PROGRAM	PUBLIC*	READ
READLN	RECORD	REPEAT
RESET	REWRITE	SET
SIZEOF*	STATIC*	SUCC
THEN	TO	TYPE
UNTIL	VAR	WHILE
WITH	WORD*	WRITE
WRITELN	XOR*	

Those reserved words which are suffixed with an asterisk are part of the language extensions of **DEFT Pascal**.

3.2 Identifiers

Identifiers are groups of letters and numbers which begin with a letter (either upper or lower case) and contain up to 12 upper or lower case letters and numbers which are not the same as any of the above listed reserved words. As in BASIC, these identifiers are used to represent variables. However, in Pascal they can also be used to represent constants, types, procedures and functions as well.

3.3 Labels

Labels are used to uniquely identify executable statements so that an executable statement may be referenced with the GOTO statement. A Pascal label functions much in the same way as line numbers do in BASIC. A label is a number which can be up to four digits long, which prefixes an executable statement with a colon (:) in between. The following is an example:

```
100: I := J
```

All labels within a block of executable statements must be declared with the LABEL declaration statement prior to the block of executable statements. The following is an example:

```
LABEL 100;
```

3.4 Constants

There are five types of constants supported by the **DEFT Pascal** Compiler. They are individually described below:

Decimal Integer Constant - A decimal integer constant is a group of numbers which may be optionally preceded with either a + or -. The allowable range for decimal integer constants is -32768 to 32767. The following are some examples:

```
-45
 45
+10234
+32768 (illegal, too large)
```

Hexadecimal Integer Constant - A hexadecimal integer constant is a group of up to 4 hexadecimal digits that is preceded with a $. A

hexadecimal digit may be any of the following: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Note that only upper case characters can be used. The range of hexadecimal integer constants is $0000 to $FFFF. The following are some examples:

```
$ABC
$12A5
$5
```

Hexadecimal integer constants are not part of standard Pascal but a form of it can be found in many Pascal implementations.

Character Constant - A character constant is a single ASCII character (other than carriage return) contained between single quotes ('). Following are some examples:

```
'A'
'a'
'&'
''''
```

The last example is a character constant that represents a single quote. The single quote is doubled.

String Constant A string constant is similar to a character constant except that more than one character is contained between the quotes. The following are some examples:

```
'PAGE HEADING TITLE'
'Sam and Joe''s Sub Shop'
```

Note that in the last example, the two single quotes in Joe''s actually is interpreted as one single quote in the string. In addition, a character constant can be used anywhere a string constant is required but the reverse is not true.

Real Constant - A real constant is a signed, decimal, fractional number, optionally raised to a signed decimal power. The general form of a real constant is:

<sign><number>.<number>E<sign><number>

The allowable range of real constants is 1E-64 to 9E63 both positive and negative. Following are some examples:

1.
-6.74
56.3E6
1.2E-3

The only required elements in a real constant are the first <number> and the decimal point (.). NOTE: Standard Pascal requires at least one decimal digit after the decimal point.

Constant Identifiers - Through the use of the CONST statement described later, identifiers can be defined as constants of some *type*. Three constant identifiers are predefined: *true*, *false* and *nil*. Later sections on *Constants*, *Types* and *Expressions and Assignments* provide more information on these constants.

3.5 Special Operators

As in BASIC the characters +, -, * and / are used as operators. However, Pascal also has several two character operators. These are as follows:

<>	**not equal**
>=	**greater or equal**
<=	**less or equal**
..	**range**
:=	**assignment**

3.6 Comments

Comments may be interspersed between (but not in the middle of) any of the above language elements. A comment starts with the characters (* and ends with the characters *). Unlike BASIC, Pascal comments can extend through more than one line. All the characters following the (* are considered comments until the *) is found later on the current or subsequent line.

4 CONST Statement

Constants as language elements are a part of practically every programming language. BASIC contains both real number and string constants. As described in the section on *Language Elements* Pascal contains decimal integer, hexadecimal integer, character, string and real constants as well as constant Identifiers.

There are two ways to create *constant identifiers*. One way is through the definition of *enumerated types* described in the section on *Types*. The other is through the use of the *const* statement. The general form of the *const* statement is as follows:

CONST <**identifier**> = <**constant**>;
.
.
.

Following are some examples:

```
CONST   MinSize = -3;
        MaxSize = 3451;
        CharLit = 'G';
        StringLit = 'This is a STRING constant';
        ExtraSize = MaxSize;
        Yes = True;
```

The purpose of the CONST statement is to allow the programmer to symbolically define a particular constant value for use later in the program. Note that any type of constant including a previously defined constant identifier may be used on the right hand side of a constant statement.

5 Types

The concept of *type* is not entirely unique to Pascal. However, the existence of a *TYPE* statement is a new concept for those programmers used to BASIC. When using BASIC, you have four kinds (types) of data: numbers, strings, number arrays and string arrays. You have different operations that can be performed with each and their internal representations are different.

A *type* refers to a data structure rather than any particular allocation of that structure. It has both a size and a set of operations that can be used on it. See the section on *Variables* for the actual allocation of memory for a given *type*.

In **DEFT Pascal**, *real numbers* and *strings* are both available along with a number of other *types, including some types that you can define yourself.* There are three classes of *types: simple, structured* and *pointer.* Those *types* which refer to indivisible entities are referred to as simple. An example is the set of whole numbers. Those which are made up of groups of simple *types* are referred to as structured. An array is an example of a structured *type.* A *pointer type* refers to those entities (such as *memory addresses)* which identify an occurrance of a *type.*

As shown in the chapter on *Program Structure* the general form of the TYPE statement is as follows:

TYPE <identifier> = <type definition>;

.
.
.

This statement causes the <identifier> to be associated with the <type definition>. Following are descriptions of all the possible *type* definitions.

5.1 Type Identifier

A previously defined *type* identifier can be used as a *type* definition. These identifiers include all those defined in previous *TYPE* statements as well as a number of *pre-defined types* that are available. These predefined *types* are as follows:

- *Integer* - This is a 16 bit (2 bytes) Ordinal *type* which can range in value from -32768 to 32767.
- *Real* - This is a 6 byte floating point number. The high-order bit of the first byte is the sign of the number. The low-order 7 bits of the

first byte is the signed exponent. The last 5 bytes contain the mantissa in the form of 10, BCD digits. The range of the exponent is 63 to -64 and reflects powers of 10.

- *Char* - This is an 8 bit (1 byte) ordinal *type* which can range in value from NUL to DEL. These are the ASCII characters with binary values from 0 to 127. In addition, the characters that correspond to the binary values from 128 to 255 are also included.
- *Boolean* - This is an 8 bit (1 byte) Ordinal *type* which can have only two possible values: 0 (false) or 1 (true).
- *String* - This is an 81 byte structured *type* which can contain a variable number of *Char types*. A minimum of 0 and a maximum of 80 Chars can be contained in a String *type*. See *Advanced Pascal* for more information on strings.
- *Text* - This is a structured *type* which defines a *FILE OF Char*. This *type* occupies 286 bytes. See the section on *Input/Output* for more information.

One additional term is that of *ordinal type*. All simple *types* except *real* are also *ordinal* types. Ordinal *types* are simple types that have explicit, discrete values.

See the section on *Expressions and Assignments* for a discussion of the kinds of operations that can be performed on these various *types*. An example of a TYPE statement using a *type* identifier:

```
TYPE Number = Integer;
```

Number is a new *type* that is fully compatible in expressions with Integer.

5.2 Enumerated

One way you can define your own *type* is by listing a set of values that are to be associated with a *type*. This defines a new ordinal *type*. The general form of an enumerated *type* definition is as follows:

```
(<identifier>, ... , <identifier>)
```

An example of a *TYPE* statement using an enumerated *type* definition is the following:

```
TYPE Color = (Red, Green, Yellow, Blue, Orange, Brown);
```

Color becomes a new independent *type* and any variables of this *type* will be protected from variables of other *types* in an expression. All enumerated *types* are 8 bit values where the identifiers contained in the list are implicitly defined as *constants* of that *type*. The order of the identifiers in the list is important. The internal representation of the first value is always 0, the second is 1 and so forth. See the section on *Expressions and Assignments* for a description of the operations that can be performed on an Enumerated *type*.

5.3 Subrange

A Subrange is a subset of values of an *Ordinal type*. The general form of a Subrange definition is as follows:

<constant>..<constant>

Where the first <constant> must be less than or equal to the second <constant>. Some examples of subrange *TYPE* statements are as follows:

```
TYPE    SmallColor = Green..Blue;
        SmallInt = -128..127;
```

Note that in the case of a subrange of integers, a subrange of -128..127 or less will result in an 8 bit *type* which is fully compatible with the full 16 bit integer *types*.

5.4 Sets

A set is a collection of specific occurrances of objects of the same type. The general form of a set definition is as follows:

SET OF <type identifier>

Where the <type identifier> specifies the types of objects comprising the set. The following is an example of use:

```
TYPE    SmallColor = (Green,Yellow,Red,Blue);
        SomeColors = SET OF SmallColor
```

SmallColor is an enumeration, and *SomeColors* is a set type. Variables of the type *SomeColors* are sets with 0 to 4 members which were listed in the declaration for the type *SmallColor*.

All Sets are 32 byte structured *types*. Each bit position within those 32 bytes represents each member of the set. Where bit 0 of byte 0 represents member 0. Bit 1 of byte 0 represents member 1, and so on

up to 255. All Sets may have up to 256 members. Sets are given values by specifying a set constant as a list of constants enclosed by []s. If a set has no values assigned, it is called an empty set, which is denoted by two empty brackets [].

```
BriteColors := [Yellow, Red];
DarkColors := [Green, Blue];
NoColors := [];
```

5.5 Arrays

An array is a familiar concept to most programmers. In Pascal, it is a list of *types* (which themselves can be arrays). The general form of an Array definition is as follows:

```
ARRAY[<ordinal type definition>] OF <type definition>;
```

where the <ordinal type definition> defines not only the *quantity* of <type definitions> in the ARRAY but also *how each element is identified by type*. The following examples should make this clear.

```
TYPE    ColorList   = ARRAY[1..6] OF Color;
        Numbers     = ARRAY[Green..Orange] OF Integer;
        Flags       = ARRAY[Color] OF Boolean;
        ColorPlane  = ARRAY[0..200] OF ColorList;
```

In the first example, a list of colors is being defined. Elements of the list are identified by the integers 1 through 6 for a total of 6 elements. Note that one of the most frequent uses of *subrange* types are in *array* definitions.

The second example shows one of the unique properties of Pascal. In this case we are defining a 4 element list of numbers where elements of the list are identified, in order, by the colors Green through Orange. The third example is similar where the number of Boolean elements is equal to the total number of colors and each element of the list is identified by a different color.

The final example shows a definition of a two-dimensional *array*. In this example there are 201 lists defined. Variables of this type would have memory organized as follows:

zeroth	**ColorList (elements 1 through 6)**
first	**ColorList (elements 1 through 6)**
second	**ColorList (elements 1 through 6)**
.	
.	
.	
two hundredth	**Colorlist (elements 1 through 6)**

Alternate (equivalent) forms of multiple dimension ARRAY declarations are as follows:

```
TYPE ColorPlane = ARRAY[0..200] OF ARRAY[1..6] OF Color;

                  or

TYPE ColorPlane = ARRAY[0..200, 1..6] OF Color;
```

Note that there is no limit to the number of dimensions allowed and that each dimension can be of a different ordinal type.

The predefined type *string* is actually an *array*[*0..80*] *OF char*. **DEFT Pascal** supports a number of language extensions associated with this *type*. See *Advanced Pascal* for language extensions on both strings and arrays.

5.6 Records

A record is a collection of data of diverse types which are located contiguously in memory in the order in which they appear in the *record*. Each data element or item is referred to as a field. A field may be of any *type*. This means that a record field may be an array, another record, a set, and so on. The general form of a record definition is as follows:

```
RECORD

<field list>

END
```

Where the <field list> has a <fixed part> and/or a <variant part>. The <fixed part> is a group of fields which are declared very much like variables. The following is an example of a RECORD with only a <fixed part>:

```
TYPE Employee = RECORD
                  Name          : String (20);
                  Street, City  : String (20);
                  State         : String (2);
                  ZipCode       : String (5);
                  Number        : Integer;
                  END;
```

In addition to a <fixed part> a RECORD can also have a <variant part> This part describes several *alternative* <field list>s which are located in the *same area of memory.* This allows you to describe the same area of memory in more than one way. The general form of the <variant part> is as follows:

```
CASE [<identifier>:] <type identifier> OF

   <constant>, ... ,<constant> : ( <field list> );
                          .
                          .
                          .
   <constant>, ... ,<constant> : ( <field list> )
```

The <identifier>: following the CASE keyword is optional and if present defines the last *fixed* field in the *record.* The <constant>s must all be of the same *type* as the <type identifier>. Each (<field list>) begins at the same position in the *record.* The size of the *record* will be determined by the size of the largest (<field list>). The following example should make things more obvious:

```
TYPE JobType = (Manager, Worker, Secretary);
     Employee = RECORD
                    (* Fixed Part Starts Here *)
                      Name : String (20);
                      Address: RECORD
                                   Street, City : String (20);
                                   State : String (2);
                                   ZipCode : String (5);
                                   END;
                      Number : Integer;
                    (* Variant Part Starts Here *)
                      CASE EmployeeType : JobType OF
                        Manager : (TotalWorkers : Integer;
                                   SecName : String (20));
                        Worker : (ManagerNbr : Integer;
                                   TotalTools : Integer;
                                   RoomNumber : Integer);
                    END;
```

In this case we have a <variant part> based on the employee's job type. The fields following the *manager* constant describe the information required for a manager. The fields following the *worker* constant describe the information required for a worker. Only one <field set> or the other will be present in any given occurance of an *employee type*.

Note that the size of *Employee* is 21 (Name) + 51 (Address) + 2 (Number) + 1 (EmployeeType) + the size of the largest variant which is the one represented by the *manager* constant (which is 23). Although not shown here, the <field lists> in the <variant part> can themselves have <variant parts>.

5.7 Pointers

A *pointer* is a reference to a specific instance of a *type*. In standard Pascal, this instance is created via the *NEW* procedure. A pointer is basically the memory address of a variable of a specific *type*. You can create a pointer type by preceding any *type* definition with an uparrow (^). The general form of a *pointer type* is:

<type definition>

An example *pointer type* definition is:

```
TYPE EmployeePtr = ^ Employee;
```

This defines a *type* called employeeptr which is a pointer to a *record type* called *employee*. you can create an instance of *employee* using the *NEW* procedure as follows:

```
NEW (EmployeePtrVar);
```

This allocates memory for an instance of *employee* and sets the memory address of that instance in the variable called *employeeptrvar* which is of type *employeeptr*.

The size of a *pointer type* is always 2 bytes regardless of the size of the *type* that it is referencing. See *Advanced Pascal* for **DEFT Pascal** extensions on the use of *pointer types*.

5.8 Files

In Pascal, both *files* and *arrays* are lists of elements. With an *array* each element can be randomly accessed. With a *file* each element can be only sequentially accessed. Files are the structured *type* that represent peripheral devices such as tape, disk, printer, keyboard and screen.

In Pascal, each element of a *file* can be of any type. File *types* other than *file of char* are used to transfer occurances of the binary image of the *type's* internal representation to and from I/O devices. A *file of char* has special (but standard Pascal) properties which provides for automatic conversion between the internal binary representation of data and the external ASCII representation. A complete explanation can be found in the section on *Input/Output*. The standard predefined type identifier *text* (file of char) can be used in *file* type declarations:

```
TYPE ThisType = FILE OF Char;
     ThatFile = Text;
```

Both of these declarations define equivalent *type* identifiers. Note that a FILE of a given *type* has a size which is equal to the size of the *type* plus 286 bytes.

5.9 PACKED Types

The reserved word PACKED may precede either *set*, *array*, *record* or *file* in a *type* declaration. In standard Pascal, this reserved word indicates that the corresponding structured *type* should be organized to occupy the least possible amount of memory. There are subsequently some restrictions on the use of these *packed types*.

With the **DEFT Pascal** Compiler, the keyword PACKED is allowed but ignored in *set*, *array*, *record* and *file* type declarations. This means that the memory requirements don't change and the restrictions are not imposed on the resulting *types*. An example of use is as follows:

```
TYPE ColorList = PACKED ARRAY[1..6] OF Color;
```

6 Variables

A variable in Pascal represents a specific memory allocation of a *type*. More important is when that memory allocation is made.

6.1 Automatic Allocation

In BASIC, a variable is allocated memory when it is first used. In assembly language a variable is allocated memory when the program is loaded into memory (provided it was declared with an RMB opcode).

In the section on *The Pascal Program* the block structure of Pascal is explained. Constants, types, procedures, functions and variables become known only when the *block* in which they are declared is activated. For variables, this also causes the memory for them to be allocated. When the block is deactivated, not only do the identifiers become unknown but the memory allocated to the variables is deallocated.

The implications of this allocation scheme are two-fold:

1. The value of any variable is undefined when the block is first activated. This is true even if the block was previously activated and deactivated. Variables will *not* assume the value that they had when the block was last deactivated.
2. An active block can activate itself causing a second allocation of its variables. Each concurrent activation of a block therefore has its own independent copy of each variable. This allows for recursive *procedures* and *functions*.

6.2 VAR Declaration

Variables are declared with the *var* statement. The general form of the statement is as follows:

```
VAR <identifier> : <type definition>;
                .
                .
                .
```

For example:

```
VAR  I : Integer;
     ThisEmployee : Employee;
```

7 Procedures and Functions

The concept of a group of statements which perform a given operation is certainly not new to a BASIC programmer. The *gosub* statement allows exactly this type of operation. In Pascal, the *procedure* statement allows a programmer to set aside a group of statements explicitly for this purpose.

In BASIC the concept of a function is provided by the *DEF FN* statement. This statement provides the ability to define single line functions. In Pascal, the *function* statement (which is almost identical to the *procedure* statement) provides a general function definition capability.

The facilities found in Pascal for defining *procedures* and *functions* are very powerful and constitute one of the major characteristics of the Pascal language. As described in the section on *The Pascal Program* Pascal is a block structured language with *procedures* (and *functions)* at the heart of this structure. It is important to read and understand this section in order to use the features of the language to their fullest.

7.1 PROCEDURE Declaration

The *procedure* statement is a declaration statement which provides the ability to construct a complete subprogram which may itself contain subordinate subprograms *(procedures* and *functions).* The general form of the declaration is as follows:

PROCEDURE <**identifier**> <**formal parameter definition**>;

<**declaration statements**>

BEGIN
<**executable statements**>
END

As mentioned in the section on *Block Structure* the <*declaration statements*> *BEGIN* <*executable statements*> *END* constitute a <block> which is *exactly* the same as a *program's* <block>. The <formal parameter definition> can be null if there are no parameters to pass to the procedure or can have the following form if parameters are present:

(<**parameter**>; <**parameter**>; ... <**parameter**>)

Where the form of <parameter> is:

VAR <identifier>, ... ,<identifier> : <type identifier>

OR

<identifier>, ... ,<identifier> : <type identifier>

The *var* keyword is present when the parameter is a *reference* parameter and is not present when the parameter is a *value* parameter. The difference between these two classes of parameters is important and is discussed in full in the next section on *Procedure Invocation*. Following are some examples:

```
PROCEDURE TestProc (VAR Parm1 : Integer; Parm2, Parm3 :
Integer);
  BEGIN
    Parm1 := Parm2 + Parm3
  END;

  PROCEDURE TestProc2;
  BEGIN
    IF GlobalVar1 > 0 THEN GlobalVar2 := 5;
    GlobalVar3 := GlobalVar1 + 3
  END;
```

You notice in the first example that *Parm1* is a reference parameter and *Parm2* and *Parm3* are value parameters. In the second example *GlobalVar1*, *GlobalVar2* and *GlobalVar3* are all variables declared outside the procedure *TestProc2*. See the section on *The Pascal Program* for a discussion of scope.

7.2 Procedure Invocation

Unlike BASIC's *gosub* statement, Pascal has no *call* statement for invoking a procedure. In Pascal, a procedure is invoked by name. That is, a *procedure* declaration implicitly defines a new executable statement which is the procedure name and is formatted according to the <parameter definition> provided in the declaration. The general form of a procedure invocation is:

<identifier> <actual parameters>

If the corresponding <formal parameter definition> in the *procedure* statement was null then the <actual parameters> must also be null. Otherwise the *actual* parameters must agree with the *formal* parameters in ordering, type and number. Some examples:

```
TestProc1 (I, 3, J*5);
TestProc2
```

Before explaining the above examples, it is necessary to define what *reference* and *value* parameters are. A formal *reference* parameter represents the actual *variable* used when the procedure is invoked. The parameter used in the procedure invocation *must* be a variable. In this case, all references to the *formal* parameter (the one in the *procedure* declaration statement) will reference the *actual* parameter (the one in the procedure invocation statement). This means that the actual parameter's value will be changed if the procedure modifies the formal parameter's value.

A formal *value* parameter represents the value of a general expression used when the procedure is invoked. In this case, any type compatible expression is allowed as the *actual* parameter since a separate allocation of memory is made when the procedure is invoked and is initialized to that value. The formal parameter in this case represents its own memory area rather than that of another variable. Changing the formal parameter in this case, does not change the value of any other variable.

In the first example above, *I* is a *reference* parameter and *3* and *J*5* are value parameters. When *TestProc1* is invoked in this case, *I* is assigned the value *3 + J*5*. Since *TestProc2* has no formal parameters, it therefore has no actual parameters.

7.3 FUNCTION Declaration

The *function* statement is almost identical to the *procedure* statement described above. This is because a *function* is a special type of *procedure* which is invoked in a different manner from a regular *procedure* and has a typed value associated with it. The syntax of the *function* statement is as follows:

```
FUNCTION <identifier> <formal parameter definition> :
                        <type identifier>;

  <declaration statements>

  BEGIN
  <executable statements>
  END
```

The only difference between the *function* statement and the *procedure* statement is the beginning keyword (*FUNCTION*

instead of *PROCEDURE)* and the presence of the *:<type identifier>* following the parameter definition. Following are some examples:

```
FUNCTION TestFunc (VAR Parm1 : Integer; Parm2, Parm3 :
Integer)
                    : Boolean;
BEGIN
  Parm1 := Parm2 + Parm3;
  TestFunc := (Parm2 > Parm3)
END;

FUNCTION TestFunc2 : Integer;
BEGIN
  IF GlobalVar1 > 0 THEN GlobalVar2 := 5;
  GlobalVar3 := GlobalVar1 + 3;
  TestFunc2 := GlobalVar3 * 2;
END;
```

You'll notice that these examples are similar to those used in the *Procedure* section except that there is an extra assignment statement at the end of each *function*. These statements use the *function* name on the left side of the assigment symbol to assign a value to be *returned* by the *function*. Every *function* is required to have at least one assigment statement which performs this task. If more than one assigment takes place, the last assigment made before the *function* terminates is the one that will be used. A function can only be of a simple type.

7.4 Function Invocation

A function is invoked by referencing its name (and supplying any required actual parameters) in an expression. In this form the *function* reference is similar to a reference to a variable. Following are some examples:

```
IF TestFunc (I, 3, J*5) THEN I := 0;
GlobalVar2 := TestFunc2 * 5
```

Note that for purposes of recursion there is no ambiguity as to whether a function is being recursively invoked or having its returned value set for its current invocation. An *invocation* occurs when the *function's* name (and actual parameter list) are found in an expression. A *function's* returned value is set when its name alone is found on the left side of an assigment statement.

7.5 FORWARD References

In Pascal, a function or procedure may be referenced by another procedure or function only if the function or procedure being referenced has been defined previous to the procedure or function making the reference. There are times when this restriction is undesireable. The *forward* declaration in Pascal solves this little problem.

A forward reference is allowed only if the procedure or function being referenced has been defined using the *forward* declaration. The following is an example:

```
    PROCEDURE TestProc (VAR Parm1 : Integer; Parm2, Parm3 :
Integer);
    FORWARD;

    PROCEDURE TestProc2;
    VAR I, K, M : Integer
    BEGIN
      K := 17; M := 23;
      IF GlobalVar1 < 0 then TestProc (K, M);
      IF GlobalVar1 > 0 THEN GlobalVar2 := 5;
      GlobalVar3 := GlobalVar1 + 3
    END;

    PROCEDURE TestProc;
    BEGIN
      Parm1 := Parm2 + Parm3
      IF Parm1 <> 40 THEN TestProc2
    END;
```

Note that TestProc has been declared as *forward* and is referenced by *TestProc2*, even though *TestProc* is defined after *TestProc2*. The same rules and conventions apply for functions as well.

8 Expressions and Assignments

Expressions are the combination of constants, variables and functions with operators to form some result. This result can then be stored (assigned) in a variable, used as a parameter to a procedure or function, used as a subscript in an array specification, used to control the execution of the program or output to a file.

8.1 Factors

The fundamental elements of an expression are called *factors*. Factors are the constants, variables and functions previously mentioned. Following are some examples of factors:

```
(* Constants *)
2
'A'
'JOES''S PLACE'

(* Variables *)
I
MyColors[1]
OurColors[137,3]
MyRecord.ItsColor

(* Functions *)
CHR (65)
ABS (-3)
```

The *value* of a factor is dependent on what kind of factor that it is. A constant has a single given value that is always used whenever that constant is referenced.

A variable's value will be potentially different each time that it is referenced. The last value that was stored (assigned) to that variable before a given reference will be the value of that variable for that reference.

In the example above you can see a reference to an *array* type variable. The value contained in the square brackets ([]) (which can be a full expression) is called a *subscript* and identifies which element of the array is being referenced. Note that every element of an array is considered to be an independent variable. When an array has more than one dimension, the subscripts are ordered according to the *type* definition for that array and are separated from each other by commas.

A reference to a field within a *record* is also a factor. This is done by naming the record, appending a period (.) and then naming the field. If the *record* is an element of an *array*, then the period follows the right bracket. For Example:

```
ArrayOfRec[i].Field1
Record1.ArrayField[i].SubField1
```

Notice that *arrays* of *records* and *records* of *arrays* can be referenced by following the above rules.

A reference to a function will actually cause the function to be invoked at the point of reference. The value returned by that invocation will be the function's value for that reference.

Another type of factor is the *inline set:*

```
(* In-Line Sets *)
[Green..Blue, Yellow]
['0'..'9', 'A'..'Z']
[1, 5, 7, 1..50]
```

An inline set is a *set* value that is built from a list of itemized ordinal expressions and subranges as shown above. Note that an inline set must always be preceeded in an expression with some indication as to what type it should assume. Therefore, it cannot be used as the first factor in a boolean expression.

A final type of factor is a *dereferenced pointer*. This is a reference to a variable whose address is in a *pointer type* variable and can be made by naming the *pointer* variable and following it with an up-arrow (^). The same syntax is used to reference the window of a *file type* variable. For example:

```
PtrVar
FileVar
```

8.2 Arithmetic Operators

An expression does not have to have any operators so that a single factor can be considered to be a full expression. However, frequently we wish to combine one or more *integer* or *real type* factors arithmetically. This is done with the use of the following operators:

+	**Addition**
-	**Subtraction**
*	**Multiplication**
/	**Real Division**
DIV	**Integer Division - quotient result**
MOD	**Integer Division - remainder result**

In addition to the above standard arithmetic operators, the **DEFT Pascal** Compiler also provides the following additional arithmetic operators:

AND	**Bitwise logical AND**
OR	**Bitwise logical inclusive OR**
XOR	**Bitwise logical exclusive OR**
LSR	**Bitwise shift right (zero fill)**
LSL	**Bitwise shift left (zero fill)**

Some examples of simple arithmetic expressions are as follows:

```
I + R          (* sum of I and R, real result *)
2 * 3          (* product of 2 and 3 *)
J / 6          (* real quotient of J divided by 6 *)
J DIV 6        (* integer quotient of J divided by 6 *)
I AND $1FF     (* value of I with high 7 bits cleared *)
J LSL 3        (* value of J shifted left 3 bit positions *)
```

8.3 Integer/Real Expressions

All the above operators (except the slash) can be used with *integer types* to create *integer* type expressions. The plus (+), minus (-), asterisk (*) and slash (/) can also be used with *real* types to create *real* type expressions.

You can also include *integer* types in *real* expressions and **DEFT Pascal** will automatically convert the *integers* to *reals*. However, you must use either the *TRUNC* or *ROUND* built-in functions to convert from *real* to *integer*. These are described in the section on *Built-In Procedures and Functions*. Following are some examples of expressions mixing *integers* and *reals*:

```
R := 1;              (* legal *)
I := 1.0;            (* illegal *)
R := I + R;          (* legal *)
IF R + I = 0 THEN ...   (* legal *)
IF I + R = 0 THEN ...   (* illegal *)
```

In **DEFT Pascal** the last expression is illegal because the expression started out as integer before the *R* was encountered. In standard Pascal, this would be a legal expression.

8.4 Arithmetic Precedence

In the above examples we saw how two factors could be combined with an arithmetic operator. In general, there is no limit to the number of factors that can be combined in a single expression. For example:

```
I * J + 5 DIV 3 OR $FF00
```

The above example is a legal expression. Unfortunately it is not immediately clear how it might be evaluated. This is because it is not clear which order the operations are performed in. In Pascal, as in most languages, this is resolved via rules of *precedence*. For arithmetic expressions the operators are divided into two categories: *multiplying* operators and *addition* operators as shown below:

Multiplying Operators: *** / DIV MOD AND XOR LSR LSL**

Addition Operators: **+ - OR**

Expressions are generally evaluated from left to right with the multiplying operations performed before the addition operations. In the example above, the evaluation would occur in the following order:

```
I * J                  (* result 1 *)
5 DIV 3                (* result 2 *)
result 1 + result 2    (* result 3 *)
result 3 OR $FF00      (* final result *)
```

Parentheses can be used to change this *default* order of operations. In fact, the above expression, although legal, is generally considered poor programming practice since it is not immediately clear how the expression is to be evaluated. *All* operations (both multiplying and addition) within a set of parentheses are performed before the result is combined with operators outside the parentheses. By inserting parentheses in the above example we can change order of evaluation as follows:

```
I * (J + 5) DIV (3 OR $FF00)
```

The parentheses have changed the order of evaluation to the following:

```
J + 5                   (* result 1 *)
I * result 1            (* result 2 *)
3 OR $FF00              (* result 3 *)
result 2 DIV result 3   (* final result *)
```

Note in the above example that the * operation takes place before the OR operation. That is due to the left-right nature of the expression evaluation. Note that parentheses may be nested to form even a different evaluation as follows:

```
I * ((J + 5) DIV (3 OR $FF00))
```

The new parentheses have changed the order of evaluation to the following:

```
J + 5                   (* result 1 *)
3 OR $FF00              (* result 2 *)
result 1 DIV result 2   (* result 3 *)
I * result 3            (* final result *)
```

Note that an expression inside a set of parentheses is actually considered a *factor* and is treated as such in all expressions.

8.5 Set Expressions

Set factors can be combined into expressions with the following operators:

+	**Union**
-	**Difference**
*	**Intersection**

As in arithmetic expressions, two set factors are combined with a single operator to produce a single set result. The above operators produce the following results:

- The Union of two sets produces a set which contains all the elements present in either the first or second set.
- The Difference of two sets produces a set which contains all the elements of the first set which are not also in the second set.
- The Intersection of two sets produces a set which contains only those elements which are in both the first and second sets.

Intersection has precedence over Union and Difference.

8.6 Boolean Expressions

A *Boolean* expression has a *true* or *false* boolean result (this is actually an 8 bit result). As in arithmetic and set expressions, boolean expressions are formed with factors and operators. The boolean operators are as follows:

NOT	**Logical NOT**	**(* Unary *)**
AND	**Logical AND**	**(* Multiplying *)**
OR	**Logical OR**	**(* Addition *)**
IN	**Set Membership**	
=	**Equals**	**(* Relational *)**
>	**Greater than**	
<	**Less than**	
>=	**Greater than or Equal**	**(* Simple Types *)**
	Containment	**(* Set Types *)**
<=	**Less than or Equal**	**(* Simple Types *)**
	Inclusion	**(* Set Types *)**
<>	**Not Equal**	

Unlike arithmetic and set expressions, boolean expressions can take *any* type factor as an argument. The only restriction is that they be combined with relational operators and that the types of both factors are the same. The *not*, *and* and *or* logical operators require boolean type factors (in order to produce a boolean result). For example:

```
BoolVar1 AND BoolVar2
Integer1 = Integer2
MyColor1 > MyColor2
```

The *in* operator is used to determine whether a given ordinal value in the range 0..255 is contained within a set of the same ordinal type. For example:

```
MyChar IN ['A'..'Z']
```

The <= and >= operators have a special meaning when applied to sets.

- Set Containment (>=) produces a *true* result if all the elements of the second set are also elements of the first set.
- Set Inclusion (<=) produces a *true* result if all the elements of the first set are also elements of the second set.

Precedence in boolean expressions is about the same as in arithmetic expressions with the following addition: after all multiplying and addition operations have been performed, a single relational or set membership operation may be performed. Note that as in arithmetic expressions, parentheses can be used to alter the order of evaluation and to break the expression down into a number of factors. The following examples illustrate this:

```
 J = I AND K <= L          (* illegal expression *)
(J = I) AND (K <= L)       (* legal expression   *)
```

Where *I, J, K* and *L* are all integer variables. The following evaluation takes place:

```
J = I                      (* boolean result 1 *)
K <= L                     (* boolean result 2 *)
result 1 AND result 2      (* final boolean result *)
```

In the following example, changing parentheses changes not only the order, but also the required intermediate expression types:

```
J = I AND (L <= K)
```

The above expression is illegal unless I and J are boolean type factors. Evaluation is as follows:

```
L <= K                     (* boolean result 1 *)
I AND result 1             (* boolean result 2 *)
J = result 2               (* final boolean result *)
```

Not only factors, but arithmetic, set and boolean expressions may be combined via relational operators as follows:

```
I*3 >= J+2
Set1 <= Set2 + Set3
(I IN [5, 6, 20..30]) = OnOffVar
(L+2)*I >= K AND $1F0
```

In the last example, the AND operator is an arithmetic operator rather than a boolean operator.

8.7 Assignment Statement

This statement is similar to that found in BASIC. The symbol of assignment is different than BASIC's to distinguish it from the equals sign. The general form is as follows:

```
<identifier> := <expression>
```

The <identifier> on the left must be a variable whose value is to be set to that of the expression on the right *after* the expression is evaluated. Following are some examples:

```
I := I * ((J + 5) DIV (3 OR $FF00))
BoolVar1 := I = J
```

In the second example, BoolVar is assigned either a True or False value depending on whether I is equal to J.

9 Compound and Control Statements

Statement execution normally starts with the statement immediately following the *BEGIN* keyword in the main program block. Execution proceeds sequentially with each subsequent statement until the *END* at the end of the main program block is reached. If any other blocks are activated in the interim, execution within that block proceeds in a similar fashion.

This section primarily describes the statements that allow you to alter this general flow of execution.

9.1 BEGIN Statement

This statement allows a programmer to include more than one statement in a place in the program where normally only one statement would be allowed. This statement does not cause any change in the order of statement execution but is frequently used in conjunction with the control statements described below which do. The following is the general form of the *BEGIN* statement:

```
BEGIN
   <executable statement> ;
                .
                .
                .
   <executable statement>
END
```

Note that the semi-colon is used to separate rather than terminate statements. Since the **DEFT Pascal** Compiler supports a *null* statement, you can put a semi-colon after the last executable statement before the *END*.

9.2 IF Statement

The *IF* statement provides the capability to execute either one of two statements based on the value of a boolean expression. Following is the general form of an *IF* statement:

```
IF <boolean expression> THEN <executable statement>
                        ELSE <executable statement>
```

If the boolean expression is *true* then the <executable statement> following the *THEN* keyword is executed otherwise the <executable statement> following the *ELSE* is executed. The *else* clause is optional and if it is not present, no statement is explicitly executed

when the boolean expression is false. In any case, after the *then* or *else* clause (if present) is executed, control falls through to the next statement following the *IF* statement. Following are some examples:

```
IF I < J THEN I := I + 1 ELSE J := J + 1;
IF J*2 = 50 THEN BEGIN
  J := 5;
  I := I*3
  END
```

The last example shows how the *BEGIN* statement can be used with the *IF* statement.

9.3 WHILE Statement

The *WHILE* statement provides the capability of *repetitively* executing a given statement while a boolean expression is *true.* This is one of Pascal's *structured looping constructs.* The general form of the *WHILE* statement is as follows:

WHILE <boolean expression> DO <executable statement>

In the WHILE statement the <boolean expression> is evaluated and if found to be *true,* the <executable statement> following the *DO* is executed and the process is repeated. This continues until the <boolean expression> is found to be *false.* At that time, the <executable statement> is not executed and control falls through to the statement following the *WHILE* statement. Note that if the <boolean expression> is *false* when the *WHILE* statement is first executed, the <executable statement> following the *DO* is not executed at all.

Normally, the <executable statement> will change the value of one or more of the variables used in the <boolean expression>. Following are some examples:

```
WHILE I < J DO I := I + 3;
WHILE J > I+3 DO BEGIN
  J := J / 3;
  I := I + 1
  END
```

9.4 REPEAT Statement

The *REPEAT* statement provides the capability of repetitively executing a given statement until a boolean expression is *false*. The general form of the *REPEAT* statement is as follows:

```
REPEAT <executable statement> ;
                    .
                    .
                    .
            <executable statement>
UNTIL <boolean expression>
```

In the *REPEAT* statement the <executable statement>s following the *REPEAT* are executed. The <boolean expression> is then evaluated and if *false* the process is repeated. This continues until the <boolean expression> is found to be *true*. At that time, control falls through to the statement following the *UNTIL*. Note that if the <boolean expression> is *true* when the *REPEAT* statement is first executed, the <executable statement>s following the *REPEAT* are still executed one time.

Normally, the <executable statement>s will change the value of one or more of the variables used in the <boolean expression>. The following are some examples:

```
REPEAT I := I + 3 UNTIL I > J;
REPEAT
  J := J / 3;
  I := I + 1
UNTIL J < I + 3
```

9.5 FOR Statement

The *FOR* statement provides the capability of repetitively executing a statement while explicitly varying an ordinal variable. The general form of the *FOR* statement is as follows:

```
FOR <assignment statement> TO <expression> DO
      <executable statement>

                    or

FOR <assignment statement> DOWNTO <expression> DO
      <executable statement>
```

In both the *TO* and *DOWNTO* versions the <assignment statement> is executed first. The ordinal variable identifier to which the assignment is made is used as the *loop counter*. The testing and varying of the loop counter is different in the *TO* and *DOWNTO* versions.

In the TO version, the following sequence is performed:

1. If the loop counter is greater than the <expression>, processing in the *FOR* loop is terminated and control falls through to the next statement following the *FOR* loop. Otherwise, the following additional steps are performed.
2. The <executable statement> (which may be a compound statement) is executed.
3. The loop counter is advanced to the next higher value (see SUCC built-in function).
4. Control goes back to the first item in this sequence.

In the DOWNTO version, the following sequence is performed:

1. If the loop counter is less than the <expression>, processing in the *FOR* loop is terminated and control falls through to the next statement following the *FOR* loop. Otherwise, the following additional steps are performed.
2. The <executable statement> (which of course may be a compound statement) is executed.
3. The loop counter is reduced to the next lower value (see PRED built-in function).
4. Control goes back to the first item in this sequence.

Normally the <executable statement> will reference the loop counter although this isn't always the case. Following are some examples:

```
FOR I := 1 TO 3 DO MyColors[i] := Red;
FOR J := 0 TO 200 DO
  FOR I := 1 TO 6 DO OurColors[J,I] := Yellow;
FOR ColorVar := Green TO Orange DO
  NumbersVar[ColorVar] := 3;
```

In the second example, the <executable statement> of the first *FOR* statement was itself a *FOR* statement. The second *FOR* loop will execute to completion (6 iterations) for each iteration of the first

FOR loop. In the last example, the loop counter is an enumerated type and is used as the subscript of an *array* type variable.

9.6 CASE Statement

The *CASE* statement provides the ability to execute one of several statements depending of the value of an ordinal expression. This ordinal expression is called a *selector*. Following is the general form of the *CASE* statement:

```
CASE <ordinal expression> OF
  <constant list> : <executable statement>;
                    .
                    .
                    .
  <constant list> : <executable statement>
  ELSE <executable statement>
  END
```

The <constant list> is a list of type compatible constants separated with commas. The <ordinal expression> is evaluated and compared with each constant sequentially in each <constant list>. If the <ordinal expression> is found to equal a constant, the comparing is stopped and the <executable statement> immediately following that particular constant is executed and control is then passed to the next statement following the *CASE* expression. If none of the constants match the <ordinal expression> and the *ELSE* clause is present, then the statement following the *ELSE* is executed.

The *ELSE* clause is a common extension found in most Pascals (sometimes as an OTHERWISE clause). It is optional, but if present must follow the last case and precede the *END*. Following is an example:

```
CASE I*5+J OF
  7,9 : J := 15;
  11,12,13,14 : BEGIN I := 3; J := 2 END;
  1 : I := J + 5
  ELSE J := 0
  END;

CASE MyColors[I] OF
  Red, Orange : MyColors[I] := Green;
  Blue : I := 3
  END
```

In both examples, you will notice at least one case which has only 1 constant in its <constant list>. In the second example, the ordinal expression is of an enumerated type.

9.7 GOTO Statement

The *GOTO* statement provides the ability to cease program execution at the point of the *GOTO* statement, and then resume program execution at the point in the program identified with the corresponding label specified in the *GOTO* statement. For those used to programming in BASIC, this feature is very familiar. The **DEFT Pascal** Compiler, however, only allows a *GOTO* to reference a *label* that is defined within the same block as the *GOTO*. The following is an example:

```
GOTO 580;
```

Where 580 is a label used to identify an executable statement within the same block as the GOTO statement.

9.8 EXIT Statement

The *EXIT* statement provides the ability to deactivate a block before coming to the block's *END* statement. The *EXIT* statement is not part of standard Pascal but a form of it is found in a number of commercially available compilers. The syntax is as follows:

```
EXIT
```

When this statement is encountered, the active block in which it is found is deactivated and no further statements within that block are executed. Note that the block being referred to is one associated with a *procedure*, *function* or *program*.

Typically, the *EXIT* statement is used in conjunction with one of the other control statements in order to conditionally continue execution within a block. *EXIT* can be used to deactivate the *program* block in which case program execution terminates and control returns to BASIC.

9.9 WITH Statement

The *WITH* statement provides the ability to reference multiple fields within the same record with one statement. One or more fields of a record can be referenced within a *WITH* statement by their field names alone provided the remaining part of the name, i.e. the record name (eventually qualified by field names), is mentioned in the *WITH* statement. The syntax is as follows:

```
WITH <variable> DO <statement>;
```

For example:

```
WITH RecordName DO Field1 := X;
```

This example is equivalent to:

```
RecordName.Field1 := X;
```

The following is another example:

```
WITH RecordName.GroupName DO
  BEGIN
    Field1 := X;
    Field2 := Y;
    Field3 := Z;
  END;
```

This example is equivalent to:

```
RecordName.GroupName.Field1 := X;
RecordName.GroupName.Field2 := Y;
RecordName.GroupName.Field3 := Z;
```

Several *WITH* statements can be nested. But since field identifiers are local to the record in which they are defined, different records can have identical field identifiers. In the case of nested *WITH*s, ownership of like field identifiers is determined by the innermost *WITH* statement. This is consistent with the Pascal rules of scope. An example of nested *WITH* is as follows:

```
WITH Record1 DO
  WITH Record2 DO
    WITH Record3 DO Field1 := X;
```

DEFT Pascal allows up to eight levels of *WITH* nesting. Also, the <variable> in a *WITH* statement cannot contain a pointer dereference or a subscripted array.

10 Input/Output

Any program is totally useless unless it can, in some way, change something external to the processor. Input/Output statements allow a program to receive outside stimulus (Input) and provide a response (Output).

With **DEFT Pascal**, the primary input statements are *reset, get, read, readln* and the builtin functions *eof* and *eoln*. The primary output statements are *rewrite, put, write, writeln* and *close*. These statements and builtin functions provide a device independent mechanism for reading data from the keyboard, cassette and disks, and for writing data to the screen, printer, cassette and disks.

10.1 File Names

The device or file (a portion of the total storage on a cassette or disk) to be used in a series of Input/Output operations is identified with a *file name*. The format of a filename is as follows:

<filename>/<ext>:<device#>

This is the same format that BASIC uses for Disk files. However, by extending the device numbers, **DEFT Pascal** also uses it for the keyboard, screen, tape and printer. The <filename> is 0 to 8 ASCII characters. The extension is 0 to 3 ASCII characters. The device numbers range from -3 to 3 with the following meanings:

-3 Keyboard/Screen
-2 Printer
-1 Cassette Tape
0 Disk drive 0
1 Disk drive 1
2 Disk drive 2
3 Disk drive 3

As can be seen, the positive device numbers corresponds to BASIC's drive numbers. The negative device numbers correspond to BASIC's device numbers with the exception that the Keyboard/Screen is -3 rather than 0.

All of the fields are optional in different circumstances. When a device number of -3 or -2 is specified, there is no need for a <filename> or <extension>. When a device number of -1 is specified, the <extension> is not used. For device numbers 0 thru 3, a default <extension> is always present depending on the program being run. When a device number is not specified, 0 is assumed. Following are some examples:

:-3	**Keyboard/Screen**
:-2	**Printer**
MYFILE:-2	**Printer (filename ignored but allowed)**
TAPEFILE:-1	**Cassette Tape File**
DISKFILE/ASM	**Assembler source file on disk drive 0**
F2NAME:1	**File is on disk drive 1, default extension used**

10.2 File Variables

Rather than giving the file name in each Input/output statement and function, a *file type variable* is used. This file type variable is initialized by a *reset* or *rewrite* statement which associates it with a file name. Other statements and functions which subsequently reference this variable then cause operations to be performed to the corresponding device or portion thereof.

A file variable has a *window* which can be read (input) or written to (output) depending on how the file variable was originally initialized (using the *reset* or *rewrite* statements). You access this window by dereferencing the file variable much like the way a pointer variable is dereferenced. The procedures and functions described below provide the ability to move data between this file window and an external device or file.

10.3 INPUT and OUTPUT File Variables

There are two predefined *file of char* (text) variables available with **DEFT Pascal**. The variable *input* is pre-initialized for access to the keyboard as though a *RESET (INPUT, ':-3')* statement (see below) had been executed before your program began. The variable *output* is pre-initialized for access to the screen as though a *REWRITE (OUTPUT, ':-3')* statement (see below) had been executed before your program began.

The existence of these two pre-defined and pre-initialized variables provides the following benefits:

1. You do not need to use *reset* or *rewrite* to initialize these variables before using them in *readln*, *writeln*, etc.
2. When using *read*, *readln*, *eoln* and *eof* you can omit the <file variable> parameter in the statement and the default file variable *input* will be used.

3. When using *writeln*, *write*, *page* and *close* you can omit the <file variable> parameter in the statement and the default file variable *output* will be used. Note that although it is permissible to use *close* with *output*, it is not necessary.

NOTE: The *input* and *output* files are actually the same file which has been specially initialized to allow both input from the keyboard and output to the screen. For this reason, it is recommended that you do not use the *reset* or *rewrite* statements with these files. When you wish to do I/O to the printer, cassette or disk, setup a separate file variable as shown in the general I/O examples further on.

10.4 Overall Example

Below is an example of a simple program that prompts at the screen for a filename to be entered and then reads that file and writes it to the printer. The filename that is entered can be any of those described above in the section on *File Names*.

```
PROGRAM CopyFile (Input, Output);

VAR InFile, OutFile : Text;
    FileName : String;
    Data : String (255);

BEGIN
  Page; (* clear the screen *)
  WRITE ('FILE NAME: ');
  READLN (FileName);

  RESET (InFile, FileName);
  REWRITE (OutFile, ':-2');

  WHILE NOT EOF (InFile) DO BEGIN
    READLN (InFile, Data);
    WRITELN (OutFile, Data);
    END;

  CLOSE (OutFile);
  END;
```

In this example, *InFile* and *OutFile* are file *variables* and ':-2' is a string constant which contains a file *name*. The *reset* statement associates the file whose name has been entered into the string variable *FileName* with the file variable *InFile* and initializes it for reading. The *rewrite* associates the printer (device number -2) with

the file variable *OutFile* and initializes it for writing.

The *while* loop causes a check for end of file on *InFile* BEFORE reading the first record. The *close* statement at the end, forces any remaining buffered data to be written. When writing to the printer or the screen it is not absolutely necessary to do the *close*, but it is recommended in case the program may be changed to output to the disk or cassette.

10.5 Lazy Keyboard Input

In order to provide an easy to use interface for the keyboard, **DEFT Pascal** incorporates the concept of *lazy keyboard input*. This involves waiting until a *read* or *readln* statement is executed before actually performing an input from the keyboard.

Standard Pascal requires that the internal buffer be prefilled so that the *eof* and *eoln* and file dereferencing operations can be performed. If this were done for keyboard input, you would have to enter data into the keyboard immediately after executing any Pascal program (before your program actually begins executing any statements). This would make it very difficult for you to synchronize your prompts (via *write* and *writeln* statements) with the corresponding inputs (via *read* and *readln* statements).

The result of the *lazy keyboard input* is that *eof* and *eoln* reflect the status as of the end of the **last** *read* or *readln* statement. For example:

```
WHILE NOT EOF DO BEGIN
  WHILE NOT EOLN DO BEGIN
    READ (X);
    WRITE (X);
    END;
  READLN;
  WRITELN;
  END;
```

If the first key that you hit is the *CLEAR* key (to indicate *eof* and *eoln* on the keyboard) the inside loop will still execute once since the prompt does not appear until the *READ (X);* statement is being executed. *X* will retain whatever value it had before the *read* unless *X* is a *char* in which case it will contain a CHR (13).

Remember, *lazy keyboard input* is only used with the keyboard. Your cassette and disk input operations are pre-buffered and conform to the Pascal standard.

10.6 CLOSE Statement

This statement is required for output files (initialized via *rewrite*) to cassette or disk in order to ensure that all data has been written to the device and the directory or trailer has been written. It may also be used for screen and printer files but has no effect. Once this statement is executed, the file variable is considered uninitialized and must be initialized again (with either *rewrite* or *reset*) in order to be used. The format of the statement is:

CLOSE (<file variable>)

As mentioned above, if <file variable> is omitted, the *output* file is assumed.

10.7 EOF Function

This is a Boolean function which specifies whether *end of file* has been reached on a particular file. This function can be used on a file of any type. Its definition is:

FUNCTION EOF (VAR FileVar : Text) : Boolean;

It can also be used as though it had no parameter and the default file *input* will be assumed. Note that *eof* can be indicated from the keyboard by terminating the last line with the *CLEAR* key instead of the *ENTER* key.

10.8 EOLN Function

This is a boolean function which specifies whether an *end of line* character is next in the window on a *file of char*. Its definition is:

FUNCTION EOLN (VAR FileVar : Text) : Boolean;

It can also be used as though it has no parameter and the default file *input* will be assumed.

10.9 FILEERROR

This is an *integer* function which returns an indication of whether a file I/O error occurred on a particular file and what the error was if it did occur. This function can be used with a file of any type. Its definition is:

FUNCTION FILEERROR (VAR FileVar : Text) : Integer

It can also be used as though it had no parameter and the default file *input* will be assumed. The *integer* return is a number from 0 to -5. The possible error numbers are as follows:

- 0, *No Error*
- -1, *End of File* - The end of a given file has been reached.
- -2, *I/O Error* - This indicates that some hardware oriented problem occurred.
- -3, *File Not Found* - The file specified was not found.
- -4, *Illegal Operation* - This indicates that you attempted a read operation on an output file or a write operation on a input file. It can also occur if you attempt to do a *reset* to the printer.
- -5, *Device Full* - While doing a *rewrite* or other write operation, the device became full.

NOTE: *eof* will return a *true* anytime *fileerror* would return a non-zero. *Fileerror* is a **DEFT Pascal** extension and is not part of standard Pascal.

10.10 GET Statement

This statement (implemented as a built-in procedure) is used to input data from cassette or disk via a *file* that was previuosly initialized with the *reset* procedure. The format of the statement is:

GET (<file variable>)

The action of this procedure is to move the file window over the next element in the file. The *get* statement cannot be used in **DEFT Pascal** with a *file of char*.

10.11 PAGE

This procedure is used to output an ASCII formfeed to the specified file. When a formfeed is output to the screen, the equivalent of BASIC's *CLEAR* is performed. When a formfeed is output to the printer, it will skip to the top of the next page. The format of the statement is:

PAGE (<file variable>)

As mentioned previously, if <file variable> is omitted, the OUTPUT file variable is assumed.

10.12 PUT Statement

This statement (implemented as a built-in procedure) is used to output data to cassette or disk via a *file* that was previously initialized with the *rewrite* procedure. The format of the statement is:

PUT (<file variable>)

The action of this procedure is to output the contents of the file window to the external device or file and then empty the window. The *put* statement cannot be used in **DEFT Pascal** with a *file of char*.

10.13 RESET and REWRITE Statements

These statements are used to initialize *file* type variables for use with subsequent Input/Output statements and functions. You can think of these statements as procedures with the following definition:

```
PROCEDURE RESET  (VAR FileVar : Text;
                  VAR Filename : String;
                  VAR DefExtension : String);

PROCEDURE REWRITE (VAR FileVar : Text;
                   VAR Filename : String;
                   VAR DefExtension : String);
```

You will only need to use one or the other of the two statements. *Reset* initializes the *File Var* for input from the specified *Filename*. When using *reset* with a disk or cassette file, a file by the name of *Filename* must already exist on that device.

Rewrite initializes the *FileVar* for output to the specified *Filename*. When using *rewrite* for output to disk, if the specified disk already has a file by the name of *Filename*, it will be deleted. A new file is then created by the name of *Filename*. On cassette, a file is created by the name of *Filename* at the current spot on the tape.

The *DefExtension* specifies the default filename extension to use if one is not included as part of the *Filename* string. This parameter is optional and if not present the default extension is blank.

10.14 READ Statement

This statement is used to input data from the keyboard, cassette or disk via a *file* that was previously initialized with the *reset* procedure. The format of the statement is:

READ ([<file variable>,] <variable>, ... ,<variable>)

As mentioned above, if <file variable> is omitted, the file *input* is assumed to be referenced.

Reading from a Typed File - When using *read* to read from a *file of <type>* where *<type>* is not *char*, the <any variable> must be of the same type as the *file*. When the *read* is executed, the size of the <type> is used to determine the number of bytes to transfer. Essentially, each *read* returns the next sequential occurance of the <type> in the *file*. For example, *READ(F, X)* is exactly the same as:

```
X := F^;
GET (F);
```

Note that *typed files* can only be used with cassette and disk.

Reading from a FILE OF Char - If the <file variable> is a *file of char* then the file is assumed to consist of a set of *lines* and the action of the *READ (F,X)* statement depends on the *type* of X. The following describes the legal *types* of *X* and the associated actions of the *read* statement:

1. *Char* - The next byte of the line is directly assigned.
2. *String* - The value of the string becomes the value of the remainder of the line. The line is truncated if necessary to fit in the string.
3. *Real* - The next group of characters delimited by blanks and/or *end of line* characters is processed by *encodereal* and the result is

stored in the variable.

4. *Integer* - The next group of characters delimited by blanks and/or *end of line* characters is processed by *encode* and the result is stored in the variable.

5. *Boolean* - The same as integer except that only the numbers 0 (for FALSE) and 1 (for TRUE) are legal. You will get unpredicable results with other values.

6. *Enumerated* - The same as integer except that only the subset of numbers 0 through 255 that apply to the given type are legal. Other values will convert to non-existent members of the type.

Some examples of use:

```
READ (IntVar, IntVar2);          (* integer from keyboard *)
READ (TapeFile, StringVar);      (* string from cassette  *)
READ (DiskFile, CharVar);        (* char from disk        *)
READ (KeyBoardFile, ColorVar)    (* enumerated from keyboard *)
```

10.15 READLN Statement

This statement is identical to the *read* statement when used with *typed files* and is almost the same when used with a *file of char* except that after all the variables are read, the window is moved past the next *end of line* character.

The *readln* statement can be used with no <variables> in order to position the file window past the next end of line character without reading a data before that point.

10.16 WRITE Statement

This statement is used to output data to the screen, printer, cassette or disk. The format of the statement is:

WRITE (<file variable>, <data>:<width>:<decimal>,...,
<data>:<width>:<decimal>)

As mentioned above, if <file variable> is omitted, the file *output* is assumed to be referenced. The action of the *write* statement depends on the type of the <file variable>.

Writing to a Typed File - If the <file variable> is *not* a *file of char* then the file is referred to as a *typed file* and the action of the *write (F,X)* statement is exactly the same as:

```
F ^ := X;
PUT (F);
```

where the *type* of the file *F* must be the same as the *type* of the variable *X*. Essentially, the current value of the variable *X* is assigned to the next element in the file *F*. The <width> and <decimal> parameters cannot be specified.

Note that *typed files* can only be used with cassette and disk.

Writing To a File of Char (Text) - If the <file variable> is a *file of char* then the action of the *write (F,X)* statement depends on the *type* of *X*. The following describes the legal *types* of *X* and the associated actions of the *write* statement:

1. *Char* - The next byte of the file is directly assigned from the contents of the character followed by <width>-1 blanks. The <decimal> parameter must not be specified.

2. *String* - The value of the string is output. If the <width> is zero or not present, the number of columns reserved will be the size of the string. If the <width> is less than the size of the string, only the first <width> characters will be output. If <width> is greater than the size of the string, blanks will be output following the string. The <decimal> must not be specified.

3. *Real* - The *decodereal* procedure is used to convert the real to the string of characters to output. The <width> specifies the size of the ASCII representation to output. If the <width> is too small to fit the number, asterisks are output to indicate overflow. The default value is 6.

   The <decimal> specifies the number of places to the right of the decimal point that should be output. If <decimal> is not present or negative, scientific notation will be used.

4. *Integer* - The *decode* procedure is used to convert the integer to the string of characters to output. The <width> parameter specifies the size of the string to output with the number right-justified within the string. If <width> is not specified it defaults to 6. The <decimal> parameter must not be specified.

5. *Boolean* - The same as integer except that only the numbers 0 (for *false)* and 1 (for *true)* are be output.

6. *Enumerated* - The same as integer except that only the subset of numbers 0 through 255 that apply to the given *type* are output.

Some examples of use:

```
WRITE (IntVar);                    (* Integer to screen          *)
WRITE (TapeFile, StringVar);       (* String to cassette         *)
WRITE (DiskFile, CharVar);         (* Char to disk               *)
WRITE (PrntrFile, ColorVar);       (* Enumerated to printer *)
WRITE (IntVar:3);                  (* 3 column output spec       *)
WRITE ('The answers are ',R*3.4:10:2,
       ' and ',S/4.2::0);          (* multiple items in one *)
WRITE (Printer, ':30, 'Centered Title');
                                   (* forcing blank padding *)
```

Since *write* does not output a carriage return at the end (as *writeln* does) it is usually used for prompting and for multiple *writes* to a single line followed by a *writeln* (see following section).

10.17 WRITELN Statement

This statement is used to perform the same operations as a *write* statement. When used with a *typed file* it is identical to the *write* statement. When used with a *file of char* (Text) it is almost the same except that after all the specified outputs have been made, a carriage return *(end of line* character) is also output. This statement also allows no <data> items at all to be specified so that only the carriage return will be output. All the examples shown for *write* also apply for *writeln*. Following are some additions:

```
WRITELN;                  (* carriage return to OUTPUT file     *)
WRITELN (DiskFile);       (* carriage return to DiskFile file *)
WRITE (CHR(13))           (* equivalent of WRITELN;             *)
```

11 Builtin Procedures and Functions

This section describes a number of predefined functions and procedures that are available with **DEFT Pascal.** Although definition statements are shown in each of the descriptions, these are purely for informational purposes and are not to be used in your program.

11.1 ABS

This is an *integer* or *real* function that returns the absolute value of the value parameter that is passed to it. The Function definition is:

```
FUNCTION ABS (Value : Integer) : Integer
                    or
FUNCTION ABS (Value : Real) : Real
```

11.2 ARCTAN

This is a *real* function which is used to compute the size of an angle whose tangent is passed to the function. The size of the angle returned by the funciton is in the form of a number of radians. The function definition is:

```
FUNCTION ARCTAN (Tangent : Real) : Real
```

11.3 CHR

This is a *character* function that returns the ASCII character for the binary value specified in the passed value parameter. The function definition is:

```
FUNCTION CHR (Value : Integer) : Char
```

This function allows you to *break* type from integer to char. See *Advanced Pascal* for a more general and structured type breaking language extension.

11.4 COS

This is a *real* function which is used to compute the cosine of an angle. The size of the angle is passed to the function in the form of a number of radians. The function definition is:

```
FUNCTION COS (Radians : Real) : Real
```

11.5 CURSOR

This is a builtin procedure that allows you to position the cursor to any of 512 positions on the screen. The upper left-hand corner is position 0. Consecutive positions proceed horizontally across the screen with the beginning of each line being a multiple of 32. The lower right-hand corner is position 511. The procedure definition is:

PROCEDURE CURSOR (Position : Integer)

Note that *position* is taken modulo 512 when used.

11.6 EXP

This is a *real* function which is used to compute the value of *e* (2.718281828) to a specific power. The function definition is:

FUNCTION EXP (Power : Real) : Real

11.7 LN

This is a *real* function which is used to compute the natural logarithm of a positive number. The function definition is:

FUNCTION LN (Number : Real) : Real

11.8 MARK

This is a *general* procedure which is used to *mark* the current state of the *heap* for use later by the *release* procedure. The procedure definition is:

PROCEDURE MARK (VAR PtrVar : Ptr)

where *PtrVar* can be a *pointer* to any *type*. Any variables allocated by the *new* procedure after saving the *heap* state in *PtrVar* will be deallocated when *release* is later called using the saved *Ptrvar*.

11.9 MEMAVAIL

This is an *integer* function which is used to determine the number of bytes of memory remaining in the *heap*. The *memavail* function is a **DEFT Pascal** extension and is not a part of standard Pascal. The function declaration is as follows:

FUNCTION MEMAVAIL : Integer;

For example:

```
WHILE MEMAVAIL >= SIZEOF (Struct) DO
  BEGIN
    NEW (Struct -Ptr);
    NumCopy := SUCC (NumCopy);
  END;
```

This example is using the value of *memavail* to determine how many copies of a data structure can be dynamically allocated. NOTE: If the *heap* available is greater than 32767 bytes, then *memavail* will return 32767.

11.10 NEW

This is a *general* procedure which is used to dynamically allocate a variable from the *heap* and assign its reference to a *pointer type variable*. The *type* of variable allocated is dependent on the *type* of the pointer. The procedure definition is:

```
PROCEDURE NEW (VAR PtrVar : Ptr)
```

where *PtrVar* can be a *pointer* to any *type*.

11.11 ODD

This is a boolean function that returns a *true* if the integer value that is passed is an odd number (not evenly divisible by 2) or a *false* if the value is even. The function definition is:

```
FUNCTION ODD (Value : Integer) : Boolean
```

11.12 ORD

This is a *general* integer function that can take *any* ordinal type expression and convert it to an integer. The internal binary value of the specified type is treated as though it were an integer. The function definition is:

```
FUNCTION ORD (Value : OrdinalType) : Integer
```

where *Value* can be any *type* of ordinal expression. See *Advanced Pascal* for a more general and structured type breaking language extension.

11.13 PRED

This is a *general* ordinal function that returns the next lower ordinal value of the same type as its argument. For *integers*, it is the equivalent of subtracting 1 from the number. For *chars*, it is the ASCII character with the next lower binary value. For *booleans*, it is only legal when the argument is *true* returning *false*. For *enumerated types*, it is the next preceding enumerated value. Using our *color type* example:

```
ColorVar := Green;
ColorVar := PRED (ColorVar)
```

ColorVar is now *Red*.

11.14 RELEASE

This is a *general* procedure which is used to deallocate variables from the *heap* which were orginally allocated via the *new* procedure. The procedure definition is:

```
PROCEDURE RELEASE (PtrVar : Ptr)
```

where *PtrVar* can be a *pointer* to any *type*. Any variables allocated by the *new* procedure after saving the *heap* state in *PtrVar* (via the *mark* procedure) will be deallocated when *release* is later called using the saved *PtrVar*.

11.15 ROUND

This is an *integer* function which is used to round a *real* value to the nearest *integer* value. The function definition is:

```
FUNCTION ROUND (VAR Value : Real) : Integer
```

11.16 SIN

This is a *real* function which is used to compute the sine of an angle. The size of the angle is passed to the function in the form of a number of radians. The function definition is:

```
FUNCTION SIN (Radians : Real) ; Real
```

11.17 SIZEOF

This is a *general* integer function which is used to compute the size (in bytes) of a variable or type. The function definition is:

```
FUNCTION SIZEOF (GenVar : AnyType) : Integer
```

NOTE: *Sizeof* is a **DEFT Pascal** extension and is not part of standard Pascal.

11.18 SQR

This is a *real* function which is used to compute the square of a number. The function definition is:

```
FUNCTION SQR (Number : Real) : Real
```

11.19 SQRT

This is a *real* function which is used to compute the squareroot of a number. The function definition is:

```
FUNCTION SQRT (Number : Real) : Real
```

11.20 SUCC

This is a *general* ordinal function that returns the next higher ordinal value of the same type as its argument. For *integers*, it is the equivalent of adding 1 from the number. For *chars*, it is the ASCII character with the next higher binary value. For *booleans*, it is only legal when the argument is *false* returning *true*. For *enumerated types*, it is the next succeeding enumerated value. Using our *color type* example:

```
ColorVar := Red;
ColorVar := SUCC (ColorVar)
```

ColorVar is now *Green*.

11.21 TRUNC

This is an *integer* function which is used to truncate a *real* value to its *integer* value. The function definition is:

```
FUNCTION TRUNC (VAR Value : Real) : Integer
```

12 DEFT vs. Standard Pascal

DEFT Pascal conforms closely to the ISO Draft Proposal for standard Pascal. The following list identifies the major areas where **DEFT Pascal** differs from the standard. In many of these areas, **DEFT Pascal** differs because it has features similar to those in **UCSD Pascal.**

- Program parameters are allowed but ignored. The predefined files INPUT and OUTPUT are always defined and opened.
- Heap management is via MARK and RELEASE rather than DISPOSE.
- Strings are variable length and implemented via the predefined type STRING(n) rather than being fixed length and implemented as PACKED ARRAY[1..n] OF CHAR.
- GOTOs may only reference LABELs within the current block.
- Lazy Keyboard Input is implemented in order to allow interactive I/O.
- RESET and REWRITE require a second parameter and may optionally take a third parameter for specifying external filenames and default extensions.
- Procedural parameters are not supported. A procedural parameter is a parameter which is itself a procedure.
- PACK and UNPACK statements are not supported. However, since **DEFT Pascal** does not have the standard restrictions on the use of PACKed variables, the major reason for the use of these statements is non-existent.
- GET and PUT cannot be used with TEXT files.
- The <record variable> in the WITH statement cannot include a subscripted array or pointer dereference.
- Conformant arrays are not supported. However, extensions to ARRAY typing provide a facility for passing actual parameters of varying numbers of elements to an individual procedure.

In addition, **DEFT Pascal** provides a number of minor and major enhancements. The minor enhancements are as follows:

- The bitwise integer operators XOR, LSL, LSR, AND and OR are supported.
- Equality comparisons between like structured types is allowed.
- I/O of enumerated types to and from TEXT files is allowed.
- An EXIT statement for prematurely returning from a procedure or function.
- FILEERROR and SIZEOF builtin functions and CURSOR builtin procedure.

A complete definition of all the major enhancements is contained in the section on *Advanced Pascal Language Extensions*.

13 Error Messsages

The **DEFT Pascal** compiler generates error messages in the source listing at those points where it detects either syntax errors or encounters I/O errors while processing a source file. The compiler prints a line of dashes followed by an up arrow and the message. The arrow indicates where the error was *detected* which is not necessarily where it *occurred*. There are a number of different error messages which are listed below:

COPY NESTING TOO DEEP

A *%C* compiler control has been found in a source file at the maximum copy depth. See *How To* for information on the *%C* compiler control.

DUPLICATE SYMBOL

The constant, type, variable, procedure or function name being defined has already been used at the current block level to define another constant, type variable, procedure or function.

EXPECTING ...

This message will have various words or symbols following it depending on what the compiler was expecting to find next in the source file. This token was not found at this point.

EXPR TYPE ERROR

This message indicates that the expression type expected at this point was not what was encountered. You may have to use a type transfer function (see *Advanced Pascal)* to let the compiler know that you want to use a different type.

FILE OPEN ERROR

The source file specified on the **DEFT Pascal** Compiler's screen, the *PASCALIB/EXT:0* file or a file specified in a *%C* compiler control resulted in an error when an open was attempted.

INVALID CONSTANT

At a point in the program where a constant was expected, a legal constant was not found.

INVALID FACTOR

While processing an expression, an invalid factor was encountered.

INVALID IDENTIFIER

At a point in the program where an identifier was expected, a legal identifier was not found.

INVALID ORDINAL TYPE

An error was detected at this point while processing an ordinal type.

INVALID SIGNED TERM

A signed term was encountered while processing an expression that was not of type integer or real.

INVALID STATEMENT

An unknown type of statement was encountered at this point in the program.

INVALID TYPE DECLARATION

An error was encountered while processing a type declaration.

INVALID TYPE IDENTIFIER

At a point in the program where a type identifier was expected, a legal type identifier was not found.

INVALID VARIABLE REFERENCE

At a point in the program where a variable identifier was expected, a legal variable identifier was not found.

LABEL ERROR

An illegal label declaration was encountered, or a label was incorrectly specified.

OBJ I/O ERROR

An I/O error was encountered while trying to open or write to the object file.

OUT OF RANGE

The explicit upper bound specified with an array type identifier was outside of the range of the original array declaration. See *Advanced Pascal* for more information.

SKIPPING TO ;

Due to a previous error, the compiler has begun skipping source code until a semicolon is encountered. This message may also indicate that a semicolon is expected at this point in the program.

SOURCE I/O ERROR

An I/O error was encountered while trying to read the current source file.

STRING CONSTANT TOO BIG

This error usually results from an unmatched quote. The maximum length of string constants is 128 bytes. The first quote is 128 bytes before the point of the error.

SYMBOL TABLE FULL

The number of symbols known at this point in the program has completely filled the available symbol table space. You must restructure your program to reduce the number of known symbols at this point in order to get it to compile within the current memory constraints.

SYNTAX ERROR

This is a catch-all error for any syntax error not explicitly covered with another error message.

UNDEFINED SYMBOL

A reference is being made to a symbol which has not been previously defined.

UNEXPECTED END

An END statement was encountered when it was not expected.

UNEXPECTED EOF

End of file on the main source file was reached before the end of the program was reached. This may be caused by a mis-matched BEGIN-END, unmatched (* or an unmatched quote (').

** UNDEFINED **

This message appears in the symbol table listing. A *procedure* was declared as *forward* but was never eventually defined, a *pointer* definition referenced a *type* identifier which was never defined or a *label* was declared but never defined.

WITH ERROR

Either the compiler's maximum *WITH* nesting level (8) was exceeded or the <record variable> portion of the statement was not a record variable or was an element of an *array* or the result of a *pointer* dereference.

DEFT Macro/6809 Assembly Language

1 Introduction

DEFT Macro/6809 is a program which reads assembly language source code and produces object code suitable for linking by **DEFT Linker**. This assembler features the following facilities:

- Motorola capatible source conventions and directives
- Builtin macro facility provides for substitution of up to 9 parameters
- Copy facility provides the ability to include several source files in a single assembly (very convenient for common equate and macro definition files)
- Object code format provides relocation, separate assembly and easy interfacing to Pascal via **DEFT Linker**

This section describes the **DEFT Macro/6809** assembly language. Readers are expected to already be familiar with the 6809 instruction set, registers and addressing modes.

2 Language Syntax

The syntax used in the **DEFT Macro/6809** is generally compatible with those found in other assemblers for the 6809.

2.1 Line Format

Assembly language source code is interpreted as a series of lines read from the source file (or *copy* files). Each line is made up of up to 4 fields. These fields are separated from each other with 1 or more blanks.

1. The *label* field is an optional field which, when present, contains an identifier that is to be defined (see below). This field is present when the first character in a line is non-blank.
2. The *opcode* field is a required field which begins with the first non-blank character following the first blank character in the line. It contains either a 6809 opcode, directive or macro and is used to control how the other fields in the line are used.
3. The *operand* field is the next field after the OPCODE field. It is a required field for some opcodes and is not present for others. Most of the discussion of language syntax describes the way that this field is used. Note that this field may contain blanks in some circumstances (see below).
4. The *comment* field is the last field in the line and consists of the remainder of the line following the operand field (or opcode field if the operand field is not present).

Note that in the listing produced by the assembler, these fields are automatically lined up on predetermined column boundaries. The use of the *label*, *opcode* and *operand* fields is more fully explained in the following sections.

2.2 Identifiers

An identifier is a name used to represent either an *absolute* or *relative* value. It is a set of up to 12 letters and numbers which must begin with a letter. Lower case letters are accepted and printed as such on the listing even though they are kept internally as upper case letters.

An identifier is defined when used in the *label* field in exactly 1 source line. The *opcode* field of that source line will determine what value is assigned to the identifier. This identifier can be used in the operand field of a source line where the identifier's value will be

used. In general, an identifier can be used as an operand even in source lines preceding the one in which it is defined. For all opcodes except *EXT*, *EXTA* and *EQU* the identifier acquires the value of the *location counter* (see below) at the point in the program where the identifier is defined.

An identifier which is defined in this manner has a *relative* value. This value is one which will be *relocated* by **DEFT Linker** into an *absolute* value when the final binary file is created. The eventual value of a relative identifier is determined by adding the location in memory where the object code is located to the relative value determined by the assembler.

An identifer may be defined with an *absolute* value by using an EQU opcode and an absolute expression in the *operand* field. See *Expression* and *EQU* for more information.

2.3 Location Counter

The *Location Counter* is a 16 bit value which is kept by the assembler that represents the number of bytes of object code produced so far. You can think of it as a relative memory address (relative to the beginning of the program). This value always starts at zero and increases in value as each source line is processed. The value of the location counter is printed at the left-hand side of the page for each line of source code printed.

The location counter is represented in the *operand* field via the symbol *.

2.4 Constants

A constant is always an *absolute* 16 bit value that is represented in some specific way. The following constants are supported by **DEFT Macro/6809**:

1. *Decimal* constants are numbers in the range from -32768 to +32767. Base 10 is the default base.
2. *Hexadecimal* constants numbers in the range from $0 to $FFFF. A hexadecimal constant is identified with a leading dollar sign ($).
3. A *Single ASCII character* constant is an ASCII character preceded with a single quote ('). The value of the resulting constant is the binary value of the ASCII character. Example: 'A

4. *Double ASCII character* constants are two ASCII characters preceded with a double quote ("). The value of the resulting constant is the binary value of the first character in the high 8 bits and the value of the second character in the low 8 bits. Example: "AB

2.5 Expressions

Identifiers and constants can be combined into *expressions* with the use of the arithmetic operators plus (+), minus (-), multiply (*) and divide (/). Expression evaluation is strictly left to right with no operator precedence. There are some restrictions on the creation of expressions:

- Relative values can not be multiplied or divided.
- You can add or subtract an absolute from a relative. This results in a relative value.
- You can subtract a relative from a relative. This results in an absolute value.
- You cannot subtract a relative from an absolute.
- You cannot add a relative to a relative.

2.6 Strings

A string is a set of 1 or more ASCII characters delimited by slashes (/) or double quotes ("). The opcodes *fcc*, *title* and *stitle* are the only ones that use strings for operands. Strings cannot be combined into expressions.

2.7 Registers

The 6809 registers are named as follows:

- *A* - high order 8 bits of the general accumulator
- *B* - low order 8 bits of the general accumulator
- *CC* - 8 bit condition code register
- *D* - 16 bit general accumulator
- *DP* - 8 bit direct page register

- *PC* or *PCR* - 16 bit program counter. Both designations result in equivalent code.
- *S* - 16 bit system stack register
- *U* - 16 bit user stack register
- *X* - 16 bit index register
- *Y* - 16 bit index register

2.8 Addressing Modes

There are a number of addressing modes which may be used with the 6809 instruction opcodes. These are used in the operand field and are as follows:

- **Inherent** - this addressing mode has no operand field. The given opcode has all the addressing information necessary to complete the instruction.
- **Immediate** - this addressing mode is designated with a leading #. The expression following the # is the object of the instruction.
- **Direct** - this addressing mode is determined by **DEFT Macro/6809** when the operand expression is absolute and its high 8 bits are equal to the value in the most recent *SETDP*.
- **Extended** - this addressing mode is determined by the assembler when the operand expression is either relative, or its absolute and the high 8 bits are not equal to the value in the most recent *setdp* instruction.
- **Relative** - this addressing mode is determined by the opcode and requires a relative expression.
- **Indexed** - this addressing mode is determined when the operand is of one of the following forms:

Zero Offset	**,<reg>**
Constant Offset	**<absolute expression>,<reg>**
Accumulator Offset	**<accumulator>,<reg>**
Auto Increment	**,<reg>++**
Auto Decrement	**,—<reg>**
Program counter relative	**<relative expression>,PCR**
	<relative expression>,PC
Indirect	**[<indexed mode>]**

- **Register-Register** - this addressing mode is determined by the opcode and requires the following form: <reg>,<reg>
- **Multi-Register** - this addressing mode is determined by the opcode and requires the following form: <reg>,...,<reg>

3 6809 Instruction Summary

The following summary lists the 6809 instruction opcodes supported by the **DEFT Macro/6809** Assembler. The first column is the assembler opcode. The second column contains the addressing modes available for this opcode. The third column is the title of the instruction.

ABX	**Inherent**	**Add B to X**
ADCA	**Immediate** **Direct** **Indexed** **Extended**	**Add Memory with Carry to A**
ADCB	**Immediate** **Direct** **Indexed** **Extended**	**Add Memory with Carry to B**
ADDA	**Immediate** **Direct** **Indexed** **Extended**	**Add Memory to A**
ADDB	**Immediate** **Direct** **Indexed** **Extended**	**Add Memory to B**
ADDD	**Immediate** **Direct** **Indexed** **Extended**	**Add Memory to D**
ANDA	**Immediate** **Direct** **Indexed** **Extended**	**AND Memory to A**
ANDB	**Immediate** **Direct** **Indexed** **Extended**	**AND Memory to B**
ANDCC	**Immediate**	**AND Memory to CC**
ASL	**Direct** **Indexed**	**Arithmetic Shift Left Memory**

	Extended	
ASLA	Inherent	Arithmetic Shift Left A
ASLB	Inherent	Arithmetic Shift Left B
ASR	Direct Indexed Extended	Arithmetic Shift Right Memory
ASRA	Inherent	Arithmetic Shift Right A
ASRB	Inherent	Arithmetic Shift Right B
BCC	Relative	Branch on Carry Clear
BCS	Relative	Branch on Carry Set
BEQ	Relative	Branch on Equal
BGE	Relative	Branch on Greater Than or Equal
BGT	Relative	Branch on Greater Than
BHI	Relative	Branch on Higher
BHS	Relative	Branch on Higher or Same
BITA	Immediate Direct Indexed Extended	Bit Test Memory with A
BITB	Immediate Direct Indexed Extended	Bit Test Memory with B
BLE	Relative	Branch on Less Than or Equal
BLO	Relative	Branch on Lower
BLS	Relative	Branch on Lower or Same
BLT	Relative	Branch on Less Than
BMI	Relative	Branch on Minus
BNE	Relative	Branch on Not Equal
BPL	Relative	Branch on Plus
BRA	Relative	Branch Always

BRN	Relative	Branch Never
BSR	Relative	Branch to Subroutine
BVC	Relative	Branch on Overflow Clear
BVS	Relative	Branch on Overflow Set
CLR	Direct Indexed Extended	Clear Memory
CLRA	Inherent	Clear A
CLRB	Inherent	Clear B
CMPA	Immediate Direct Indexed Extended	Compare Memory from A
CMPB	Immediate Direct Indexed Extended	Compare Memory from B
CMPD	Immediate Direct Indexed Extended	Compare Memory from D
CMPS	Immediate Direct Indexed Extended	Compare Memory from S
CMPU	Immediate Direct Indexed Extended	Compare Memory from U
CMPX	Immediate Direct Indexed Extended	Compare Memory from X
CMPY	Immediate Direct	Compare Memory from Y

	Indexed Extended	
COM	Direct Indexed Extended	Complement Memory
COMA	Inherent	Complement A
COMB	Inherent	Complement B
CWAI	Immediate	Mask CC and Wait for Interrupt
DAA	Inherent	Decimal Adjust A
DEC	Direct Indexed Extended	Decrement Memory
DECA	Inherent	Decrement A
DECB	Inherent	Decrement B
EORA	Immediate Direct Indexed Extended	Exclusive Or Memory with A
EORB	Immediate Direct Indexed Extended	Exclusive Or Memory with B
EXG	Reg-Reg	Exhange Registers
INC	Direct Indexed Extended	Increment Memory
INCA	Inherent	Increment A
INCB	Inherent	Increment B
JMP	Direct Indexed Extended	Jump
JSR	Direct Indexed Extended	Jump to Subroutine

LBCC	Relative	Long Branch on Carry Clear
LBCS	Relative	Long Branch on Carry Set
LBEQ	Relative	Long Branch on Equal
LBGE	Relative	Long Branch on Greater Than or Equal
LBGT	Relative	Long Branch on Greater Than
LBHI	Relative	Long Branch on Higher
LBHS	Relative	Long Branch on Higher or Same
LBLE	Relative	Long Branch on Less Than or Equal
LBLO	Relative	Long Branch on Lower
LBLS	Relative	Long Branch on Lower or Same
LBLT	Relative	Long Branch on Less Than
LBMI	Relative	Long Branch on Minus
LBNE	Relative	Long Branch on Not Equal
LBPL	Relative	Long Branch on Plus
LBRA	Relative	Long Branch Always
LBRN	Relative	Long Branch Never
LBSR	Relative	Long Branch to Subroutine
LBVC	Relative	Long Branch on Overflow Clear
LBVS	Relative	Long Branch on Overflow Set
LDA	Immediate Direct Indexed Extended	Load Memory into A
LDB	Immediate Direct Indexed Extended	Load Memory into B
LDD	Immediate Direct Indexed Extended	Load Memory into D

LDS	Immediate Direct Indexed Extended	Load Memory into S
LDU	Immediate Direct Indexed Extended	Load Memory into U
LDX	Immediate Direct Indexed Extended	Load Memory into X
LDY	Immediate Direct Indexed Extended	Load Memory into Y
LEAS	Indexed	Load S with Effective Address
LEAU	Indexed	Load U with Effective Address
LEAX	Indexed	Load X with Effective Address
LEAY	Indexed	Load Y with Effective Address
LSL	Direct Indexed Extended	Logical Shift Left Memory
LSLA	Inherent	Logical Shift Left A
LSLB	Inherent	Logical Shift Left B
LSR	Direct Indexed Extended	Logical Shift Right Memory
LSRA	Inherent	Logical Shift Right A
LSRB	Inherent	Logical Shift Right B
MUL	Inherent	Multiply
NEG	Direct Indexed Extended	Negate Memory

NEGA	Inherent	Negate A
NEGB	Inherent	Negate B
NOP	Inherent	No Operation
ORA	Immediate Direct Indexed Extended	Inclusive Or Memory with A
ORB	Immediate Direct Indexed Extended	Inclusive Or Memory with B
ORCC	Immediate	Inclusive Or Memory with CC
PSHS	Multi-Reg	Push Registers on System Stack
PSHU	Multi-Reg	Push Registers on User Stack
PULS	Multi-Reg	Pull Registers from System Stack
PULU	Multi-Reg	Pull Registers from User Stack
ROL	Direct Indexed Extended	Rotate Left Memory
ROLA	Inherent	Rotate Left A
ROLB	Inherent	Rotate Left B
ROR	Direct Indexed Extended	Rotate Right Memory
RORA	Inherent	Rotate Right A
RORB	Inherent	Rotate Right B
RTI	Inherent	Return from Interrupt
RTS	Inherent	Return from Subroutine
SBCA	Immediate Direct Indexed Extended	Subtract Memory with Borrow from A

SBCB	Immediate Direct Indexed Extended	Subtract Memory with Borrow from B
SEX	Inherent	Sign Extend B into A
STA	Immediate Direct Indexed Extended	Store Memory from A
STB	Immediate Direct Indexed Extended	Store Memory from B
STD	Immediate Direct Indexed Extended	Store Memory from D
STS	Immediate Direct Indexed Extended	Store Memory from S
STU	Immediate Direct Indexed Extended	Store Memory from U
STX	Immediate Direct Indexed Extended	Store Memory from X
STY	Immediate Direct Indexed Extended	Store Memory from Y
SUBA	Immediate Direct Indexed Extended	Subtract Memory from A

SUBB	Immediate Direct Indexed Extended	Subtract Memory from B
SUBD	Immediate Direct Indexed Extended	Subtract Memory from D
SWI	Inherent	Software Interrupt 1
SWI2	Inherent	Software Interrupt 2
SWI3	Inherent	Software Interrupt 3
SYNC	Inherent	Synchronize to Interrupt
TFR	Reg-Reg	Transfer Register to Register
TST	Direct Indexed Extended	Test Memory
TSTA	Inherent	Test A
TSTB	Inherent	Test B

4 General Directives

In addition to the opcodes listed in the preceding section, which translate directly into 6809 opcodes, **DEFT Macro/6809** contains a number of directives which provide memory initialization, reservation and assembly control.

4.1 COPY

This directive allows you to copy source lines from another file into the current assembly. The standard file name found in the *operand* field is openned (with a default suffix of ASM) and read to end of file. In the current version of the assembler, files that have been copied cannot themselves contain COPY directives. Example:

```
COPY RECRDEQU:1
COPY MYMACROS
```

4.2 END

This directive is provided in order to allow the programmer to terminate his program with an END. The END directive has no OPERAND and does not result in code generation. The assembler will continue processing any source lines following an END. The assembler does not require a program to have an END since source lines are fetched until end of file is reached. Example:

```
END
```

4.3 EQU

This directive provides the capability of defining an identifier to have a specific value. The identifier found in the *label* field is assigned the value of the expression found in the *operand* field. Example:

```
LABEL1   EQU      $50
LABEL2   EQU      LABEL1*3
LABEL3   EQU      *-EARLIERLABEL
```

4.4 FCC

This directive creates an ASCII string of characters. The *operand* field contains the ASCII string to be created enclosed in either slashes (/) or double quotes ("). Example:

```
NAME    FCC   /John Q. Smith/
NAME2   FCC   "Mary Jones/MD"
```

4.5 FCB

This directive creates individual bytes with the values of the expression(s) found in the operand field. More than one byte may be defined by separating the expressions with commas (,). Example:

```
BYTES   FCB  6,$F,LABEL1,'A
        FCB  *-BYTES
```

4.6 FDB

This directive creates individual words with the values of the expression(s) found in the operand field (high bits in low order byte). More than one word may be defined by separating the expressions with commas (,). Example:

```
WORDS   FDB  5,6
        FDB  WORDS+3
```

4.7 MAIN

This directive tells the **DEFT Macro/6809** (and subsequently **DEFT Linker**) where execution should begin. Only one *main* directive should be included in a set of modules to be linked together. *Main* has no operand, so execution will be at the value of the location counter. Example:

```
        MAIN
START   ...
```

4.8 RMB

This directive reserves memory which is preinitialized to zero. The absolute expression found in the *operand* field specifies the number of bytes of memory to reserve. Example:

```
WRKAREA RMB  $200
```

4.9 SETDP

This directive specifies to the assembler what page number should constitute the direct page. This directive should generally follow a TFR instruction that loads the DP register with a new value. The expression (evaluated at assembly time) in the *operand* field is used as the new direct page number. If no *setdp* directive is given, page number 0 is assumed to be the direct page. Example:

```
LDA     #DATATABLE/256    A=Page Number
TFR     A,DP              Put in DP
SETDP   #DATATABLE/256    Tell assembler
```

Note that the above example works only if *DATATABLE* is an absolute.

5 Macros

When writing a program in assembly language, you frequently encounter situations where a group of instructions is repeated throughout your program with only minor variations. Subroutine calls which require parameters to be setup are a typical example. In this case, only the arguments are different. What is needed is a way to define a template of instructions which could then be invoked at those points in the program where they are needed. *Macros* are exactly these templates.

5.1 General Operation

Before a macro can be used it must first be defined. This definition *must* be processed by the assembler on source lines that are read before the source lines on which the macro is used. This definition includes the *name* of the macro, a *body* which includes the fixed elements as well as where parameters are to be substituted and finally an *ending* which tells the assembler that the definition is complete.

Once the definition is completed the macro may be used. The *name* of the macro used in the *opcode* field of a subsequent source line is what actually invokes the macro. The substitution parameters are placed in the *operand* field separated with commas. The previous macro definition now is included at this point in the program very much like a copy. The main difference is that the parameters included on the invocation line are substituted throughout the source lines as defined in the template. These lines are then assembled and optionally listed.

5.2 Macro Definition

A macro is defined with a *MACRO* directive. The operand field *must* contain a macro name of up to 6 characters. Note that the assembler can distinguish between an identifier and a macro with the same name. However, the assembler *cannot* distinguish between a macro and a predefined assembler opcode or directive of the same name. Following the *MACRO* directive is a number of source lines that constitute the template. The definition ends with an *ENDM* directive which has no operand field. Up to 30 macros may be defined whose templates use no more than a total of 1.5K bytes of memory.

Substitution parameters are indicated in the template lines with the percent sign (%) followed by single digit number (0 through 9). Up to

9 parameters numbered from 1 to 9 can be used. The zeroth parameter is a macro expansion count which is automatically kept by the assembler. Use of this parameter can guarantee a unique value for each expansion of the macro. This is useful when including LABEL fields in the template. Example:

```
          MACRO PASFUN
          LDD   STATICBASE,PCR
          LEAY  %1,PCR
          LEAX  %2,PCR
          PSHS  U,Y,X,D
          LBSR  PASFUN
          LEAS  6,S
          LDD   ,S++
          BEQ   PASFUN%0
          ADDD  #0%3
PASFUN%0  EQU   *
          ENDM
```

The above example generates a postion-independent call sequence to a Pascal function. The function requires two parameters whose addresses are loaded into the Y and X registers respectively. The D register is always loaded with a given base value. The PSHS sets up the stack with the U register push used only to reserve space for the value returned by the function.

On return, the macro cleans up the stack and gets the returned value in the D register via a LDD rather than a PULS in order to set the CC register. A check then follows which adds a third macro parameter to the result if a non-zero was returned by the function. Note that the third macro parameter is optional in that if it is not present, a zero is added to the result. This macro also makes use of the macro expansion counter to create an identifer for the macro's own use.

Note that the macro definition itself does not result in *any* code generation. Neither does the assembler try to parse the template so assembly errors may occur when the macro is invoked.

5.3 Macro Invocation

A macro is invoked by using its name in the *opcode* field of a source line that follows the definition. Up to 9 parameters may be included in the *operand* field separated with commas. Not all parameters need be included in a given invocation. All parameters following the

last one specified, as well as those that are explicitly not included via placeholder commas, are assigned a null value.

The previous macro defintion is then included at this point in the program. The parameters included (or not included as the case may be) on the invocation line are substituted throughout the source lines as defined in the template. These lines are then assembled and optionally listed.

An example invocation of the macro defined above follows:

```
                PASFUN STRING,COUNT,7
+               LDD    STATICBASE,PCR
+               LEAY   STRING,PCR
+               LEAX   COUNT,PCR
+               PSHS   U,Y,X,D
+               LBSR   PASFUN
+               LEAS   6,S
+               LDD    ,S++
+               BEQ    PASFUN1
+               ADDD   #07
+PASFUN1        EQU    *
```

In the above example, the three arguments are substituted where the %1, %2 and %3 are found in the template. The macro expansion count is 1 and is substituted where %0 is found in the template. The following example shows a second invocation of the same macro:

```
                PASFUN STRING1,COUNT+2
+               LDD    STATICBASE,PCR
+               LEAY   STRING1,PCR
+               LEAX   COUNT+2,PCR
+               PSHS   U,Y,X,D
+               LBSR   PASFUN
+               LEAS   6,S
+               LDD    ,S++
+               BEQ    PASFUN2
+               ADDD   #0
+PASFUN2        EQU    *
```

Notice that the third parameter was not included on the invocation line. Since the macro was constructed with a leading zero before the %3 in the ADDD line, its presence is not required for an error-free assembly. In addition, %0 was substituted with a 2 instead of a 1 providing a unique label for both macro expansions. Note that in

this macro a unique label is not absolutely required since the length of the branch is always the same and could be indicated by *+5.

6 Linkage Directives

DEFT Macro/6809 provides directives that allow the object code produced by one assembly to be combined with that of others. The primary uses of this separate assembly facility are:

- A very large program can be divided into managable pieces which are then individually coded and assembled.
- A frequently used routine or set of routines can be written and tested once. Any programs that subsequently need these routines can merely reference them and then include them with the **DEFT Linker**.
- Assembly language programs can be easily combined with PASCAL programs.

When talking about separate assembly the term *module* is used to refer to the code that is assembled via one execution of the assembler. Linking these modules together is accomplished by declaring a given identifier as *public* in the module in which it is defined. Other modules which wish to use the routine or data area defined with this identifier declare it as *external*. **DEFT Linker** then inserts the correct absolute address or offset into the code when the final binary image is created.

6.1 PUBLIC

The *public* directive is used to declare an identifier as public. The identifier must be defined elsewhere in the same module. The *operand* field contains the identifier that is to be declared *public*. Example:

```
          PUBLIC    MYSUBR
            .
            .
            .
MYSUBR    PSHS      Y,X,D   Save Registers
            .
            .
            .
```

As with any other reference to an identifer, a *public* directive can come either before or after the identifier is defined. Note that an identifier which is defined as *ext* or *exta* cannot be given the *public* attribute.

6.2 EXT and EXTA

The *ext* and *exta* directives are used to define an identifier and to declare it as *external* to this module. *Ext* defines a relative identifier. *Exta* defines an absolute identifier. The distinction does not affect the code that is generated by the assembler, but it does allow the assembler to correctly flag PIC and non-PIC code. The identifier to be defined is in the *label* field. Example:

```
YOURSUBR    EXT
YOURCONST   EXTA
            .
            .
            .
            LDD     #YOURCONST
            LBSR    YOURSUBR
```

6.3 STACK

This directive allows you to specify how much stack space this module will require at execution time. This is convenient when linking assembly language with **DEFT Pascal** so that a total stack requirement can be determined by **DEFT Linker**. If this directive is not present, the assembler assumes a zero stack requirement. The absolute expression in the *operand* field is the amount. Example:

```
            STACK  $20
```

7 Listing Control Directives

This section describes the assembler directives available to control the source listing produced by the assembler. Although these directives control the source listing, they are not included in the listing themselves.

7.1 EJECT

The assembler normally prints 55 source lines on a page before starting a new page. This directive specifies that the next source line should begin at the top of a page. There is no *operand* field. Example:

```
EJECT
```

7.2 LIST

This causes the assembler *list level* to be incremented by one. The *list level* is a value that starts at zero and determines whether source lines should be included in the list file. When this value is greater than or equal to zero, lines are included. When it is negative, source lines are not included. If a previous NOLIST (see below) made the *list level* go negative, then this directive will cause the listing to be turned back on. If the *list level* is already zero, this LIST will cancel the next following NOLIST. This directive has no *operand*. Example:

```
LIST
```

7.3 NOLIST

This causes the assembler *list level* to be decremented by one. See *LIST* for a description of the *list level*. Its general purpose is to prevent source lines from being listed. This directive has no *operand*. Example:

```
NOLIST
```

7.4 MLST

This directive causes macro expansions to be listed. Unlike the *LIST* directive, the *MLST* directive does not have a *level*. It is either on or off. This directive has no *operand*. Example:

```
MLST
```

7.5 NOMLST

This directive suppresses macro expansions from being listed. This directive has no *operand.* Example:

```
NOMLST
```

7.6 SKIP

This directive causes 1 or more blank lines to be included in the source listing. The number of blank lines included is the absolute expression in the *operand* field. If the *operand* field is not present or the expression is less than 1, then a value of 1 is used. Example:

```
SKIP    2
SKIP
```

7.7 STITLE

This directive specifies the string that is to be the subtitle string printed at the top of the listing starting with the next page. The string found in the *operand* field is used. This directive does an implicit *EJECT.* Example:

```
STITLE /Important Subroutine Name/
```

7.8 TITLE

This directive specifies the string that is to be the title string printed at the top of the listing starting with the next page. The string found in the *operand* field is used. This directive does an implicit *EJECT.* Example:

```
TITLE /Important Program Name/
```

8 Error Messages

The **DEFT Macro/6809** Assembler generates error messages in the source listing at those points where it detects either syntax errors or encounters I/O errors while processing a source file. Error messages are distinguished by the *** *ERROR* - at the beginning of the line and follow the line that they are referencing. Following are the error messages and a short explanation of each.

ADDR MODE

An invalid addressing mode was used.

BAD OPCODE

An unknown opcode or macro was used.

BAD RMB

An RMB instruction must have a positive absolute expression for an *operand.*

COPY NEST

A copied file may not have a COPY instruction in it.

DUPL MACRO

There is already a macro defined with this name.

DUPL SYMBL

There is already a symbol defined with this name.

PUBLC->EXT

An external symbol is being declared as public. This is illegal.

EXPRESSION

An illegal expression has been detected.

LABEL RQ'D

This opcode requires a symbol in the label field and there is none.

MAC SPACE

This macro definition exhausts all the available macro space and so is rejected.

MACRO NEST

You cannot invoke a macro from within a macro.

OPRND RQ'D

This opcode requires an *operand* and there is none.

OPRND SIZE

This opcode requires an 8 bit *operand* and the one that is present requires 16 bits.

PHASE

This label is being assigned a different value on the assembler's second pass than it recieved on the first pass. This is usually due to using a symbol in an RMB statement before the symbol is defined.

REGISTER

An unknown or illegal register has been specified.

UNDEF SYM

An unknown or illegal symbol has been used.

Advanced Pascal Language Extensions

Adv

1 Introduction

This section describes a number of facilities in the **DEFT Pascal** Compiler which are not found in standard Pascal. These facilities provide the programmer with significant additional capabilities which allow easier text processing, ROM and absolute memory access, and separate compilation with both **DEFT Pascal** and **DEFT Macro/6809** assembly language.

Before deciding whether to use these facilities, the purpose of the program to be written must be considered. If portability is essential then only those facilities described in Pascal should be used. If the program is to run only on the Color Computer and you wish to take maximum advantage of the machine's capabilities, then by all means use the *Advanced Pascal* features.

Note that even when using these advanced features the resulting program may still be moved to other machines since many other Pascals have corresponding features. This is especially true in the areas of string handling, separate compilation and compiler controls.

2 Strings

In standard Pascal, a *string* is little more than an *array of char*. **DEFT Pascal** allows you to treat a *string* in exactly the same way. However, a *string* is not exactly the same as an *array of char* in that you can also treat this type as a true variable length structure. This allows you to access individual elements of the string by including a subscript or access the entire structure by not including a subscript. Note that since *array of string* is allowed, the number of subscripts determine the type of the resulting factor.

A string, in **DEFT Pascal**, contains a string length in element 0. The remaining elements are the string itself. The default maximum length of a string is 80. Other maximums can be declared (up to 255) by including a constant in parentheses following the type identifier *STRING*. See the section on *Type Extensions* for a complete explanation.

Note that this structure is maintained in string constants as well as string variables.

2.1 String Assignments

The assignment statement not only allows you to assign a string variable or constant to a string, but also a general string expression. The syntax of a string assignment statement is as follows:

<string variable> := <string term> + ... + <string term>

Where <string variable> is a simple variable, record member, array element or dereferenced pointer variable with a base type of string. <string term> is any of the following:

- A *string* variable
- A *string* constant
- A *char* type expression

The result of the assignment is to set the <string variable> on the left of the assignment sign to the ordered concatenation of the <string term>s on the right side. Some examples:

```
StringVar := OtherString + ' suffix string';
StringVar := 'First line' + CHR(13) + 'Second line';
StringVar := StringVar + 'A'
```

The last example shows how to append <string term>s to the end of an existing value in a <string variable>.

2.2 String Relations

As mentioned in *Pascal*, strings may be combined with relational operators in boolean expressions. When comparing two strings, **DEFT Pascal** generates code that compares the strings on a character by character basis from left to right. When two characters that are not equal, or the end of a string is encountered the compare stops. If unequal characters are found, the binary value of the corresponding characters determines the result. If the end of a string is encountered, the longer string is considered greater. Only if the current length and all corresponding characters are equal are the strings themselves considered equal.

2.3 STRINGCOPY Procedure

This predefined procedure is used to copy a portion of one string into another. The procedure declaration is:

```
PROCEDURE STRINGCOPY (VAR SOURCE : STRING;
                      INDEX, LENGTH : INTEGER;
                      VAR DESTINATION : STRING);
```

The *string* variable DESTINATION is set to the *string* contained in SOURCE starting with INDEXth character and continuing for LENGTH characters. If the length of SOURCE is less than INDEX then DESTINATION will be null. If the length of SOURCE is less than INDEX+LENGTH-1 then the length of DESTINATION will be the length of SOURCE less INDEX-1.

2.4 STRINGDELETE Procedure

This predefined procedure is used to delete a portion of a *string* variable. The procedure declaration is:

```
PROCEDURE STRINGDELETE (VAR SOURCE : STRING;
                        INDEX, LENGTH : INTEGER);
```

The *string* variable SOURCE has the *string* starting at the INDEXth character and continuing for LENGTH characters removed from it. If the length of SOURCE is less than INDEX then no change is made. If the length of SOURCE is less than INDEX+LENGTH 1 then all the characters in SOURCE following the INDEXth character will be deleted and the new *string* length will be INDEX-1.

2.5 STRINGINSERT Procedure

This predefined procedure is used to insert one *string* into into another at a specified point. The procedure declaration is:

```
PROCEDURE STRINGINSERT (VAR SOURCE : STRING;
                        VAR DESTINATION : STRING;
                        INDEX : INTEGER);
```

The *string* variable SOURCE is inserted into the *string* DESTINATION starting in front of the INDEXth character. If the length of DESTINATION is less than INDEX then SOURCE is appended to DESTINATION.

2.6 STRINGPOS Function

This predefined function is used to find the location of one *string* within another. The function declaration is:

```
FUNCTION STRINGPOS (VAR IMAGE, TARGET : STRING) : INTEGER;
```

A search of *string* TARGET is made to try to find *string* IMAGE. If IMAGE is found in TARGET then STRINGPOS returns the character position in TARGET where IMAGE was found. If IMAGE is not found in TARGET, STRINGPOS returns a zero.

2.7 ENCODE Function

This predefined function is used to convert a *string* containing an integer constant to an integer. The function declaration is:

```
FUNCTION ENCODE (VAR ASCII : STRING) : INTEGER;
```

The *string* ASCII is scanned and the binary representation of the ASCII characters is returned. The following rules are used during the scan:

1. Leading blanks are ignored
2. A leading + or - sign is allowed
3. The scan stops when the end of the *string* or a non-numeric character is encountered

If no numeric characters are encountered before the scan stops, ENCODE returns zero.

2.8 ENCODEREAL Function

This predefined function is used to convert a *string* containing a real constant to a real. The function declaration is:

```
FUNCTION ENCODEREAL (VAR ASCII : STRING) : REAL;
```

The *string* ASCII is scanned and the binary representation of the ASCII characters is returned. The following rules are used during the scan:

1. Leading blanks are ignored
2. A leading + or - sign is allowed
3. The first set of digits are the mantissa and may contain an imbedded decimal point.
4. The letter *E* may follow the mantissa to indicate that an exponent follows.
5. The exponent may have a leading sign but cannot have an imbedded decimal point.
6. The scan stops when the end of the *string* or a non-numeric character is encountered

If no numeric characters are encountered before the scan stops, ENCODEREAL returns zero.

2.9 DECODE Procedure

This predefined procedure is used to construct a *string* containing the external representation (base 10) of an integer. The procedure declaration is:

```
PROCEDURE DECODE (NUMBER, SIZE : INTEGER;
                  VAR ASCII : STRING);
```

The *string* ASCII is constructed. NUMBER is the binary value to use during the conversion and SIZE is the resulting *string* length of ASCII. The external (base 10) representation of NUMBER is right justified in ASCII. If SIZE is larger than required, leading blanks are appended on the left. If SIZE is too small, the leftmost characters are truncated.

2.10 DECODEREAL Procedure

This predefined procedure is used to construct a *string* containing the external ASCII representation (decimal or scientific) of a real The procedure declaration is:

```
PROCEDURE DECODEREAL (NUMBER : REAL;
                      SIZE, FRACTION : INTEGER;
                      VAR ASCII : STRING);
```

The *string* ASCII is constructed. NUMBER is the binary value to use during the conversion, SIZE is the resulting total *string* length of ASCII and FRACTION is the number of fractional digits to the right of the decimal point. The external (base 10) representation of NUMBER is right justified in ASCII. If SIZE is larger than required, leading blanks are appended on the left. If SIZE is too small, the string is filled with asterisks. If FRACTION is negative, then scientific notation is used, otherwise a decimal display is used.

2.11 HEX Procedure

This predefined procedure is used to construct a *string* containing the ASCII hex representation of a specified area of memory. The procedure declaration is:

```
PROCEDURE HEX    (ADDRESS : INTEGER;
                  BYTECOUNT : INTEGER;
                  VAR ASCII : STRING);
```

The memory area beginning at ADDRESS and continuing for BYTECOUNT bytes is converted to a hex *string* which is placed in ASCII. The hex representation is a pair of hex digits followed by a blank for each byte except the last. The resulting length of ASCII is (BYTECOUNT*3)-1.

3 Type Extensions

A strongly typed language like Pascal can help a programmer gain and maintain control of his program. He can ensure that variables of different types are not inadvertantly combined in an expression or the wrong type expression is passed as a parameter to a *procedure* or *function*.

However, there are occasions when a programmer wants to treat some datum as *usually* of one type and *sometimes* to treat it as another type. The extensions pertaining to type found in **DEFT Pascal** provide a sorely needed type breaking function that is only partially found in standard Pascal.

3.1 Type Conversions

Provided in standard Pascal are the type conversion functions *chr*, *odd* and *ord*. **DEFT Pascal** supports these functions, but also provides a more regular *type* breaking capability. This capability is implemented with implicit builtin function definitions based on ordinal *type* definitions.

When any ordinal *type* is defined, **DEFT Pascal** also implicitly defines a conversion function with the same name as the *type*. This function has a value parameter which is of any ordinal *type*. It returns (in the same way that *chr* and *ord* do) the equivalent value with a *type* equal to the named *type* identifier. For example:

```
.
.
.
TYPE Color = (Red, Green, yellow);
     Fruit = (Apple, Lime, Lemon);
VAR ColorVar : Color;
    FruitVar : Fruit;
.
.
.
FruitVar := Fruit (ColorVar);
.
.
.
```

In the above example, *Colorvar* produces an expression of type *Color*. This expression is used as a parameter to the function *Fruit* (implicitly declared in the *type* definition) which converts it to a

Fruit type expression. Operation of the assignment statement is to set *FruitVar* equal to the *fruit* whose corresponding *color* is in *ColorVar*.

Note that as a result of this extension, the builtin function *integer* is equivalent to *ord* and *char* is equivalent to *chr*.

3.2 Pointer/Integer Conversions

In order to allow full use of the addressing capability of the 6809, **DEFT Pascal** provides the ability to convert between *integer* and *pointer types*. The builtin function *ptr* will convert an *integer type* to a *pointer type*. In addition, a *pointer* can be converted to an integer via the *ord* and *integer* builtin functions. These facilities make it possible to manipulate *pointers* arithmetically. For example:

```
TYPE BigRecord = RECORD ... END;
VAR BigPtr : ^BigRecord;
  ...
BEGIN
  ...
  BigPtr := PTR (ORD (BigPtr) + SIZEOF (BigRecord));
  ...
```

In the above example, *BigPtr* is incremented to point to the next *BigRecord* in memory.

3.3 Arrays

In standard Pascal an *array type* definition includes both the upper and lower bounds of the *array* as well as the element *type*. This of course is also true with **DEFT Pascal**. However, when using a previously defined array *type* identifier, you may specify a different upper bound than the default contained in the original *type* declaration. Example:

```
TYPE MyArray = ARRAY[1..200] OF Integer;
VAR Array1 : MyArray;
    Array2 : MyArray(150);
```

In the above example, *Array1* and *Array2* are equivalent *types*. However, *Array1* has 200 elements and *Array2* has 150 elements. This variable size capability is useful when creating *procedures* and *functions* which process *arrays* of a given *type* but with varying sizes. However, for all arrays except *strings*, the new upper bound

must be less than or equal to the upper bound of the original array. Standard Pascal has a *conformant array* facility which provides an equivalent capability when used in *procedures* and *functions*.

Note that since the type *string* can be used as an *array of char* type, you can also specify an upper bound (up to 255) when declaring *strings*. This upper bound will determine the amount of memory reserved for the *string* variable and the maximum length *string* value that can be stored.

4 Absolute Memory Access

This section describes the **DEFT Pascal** Compiler's facilities for accessing specific areas of the 6809 address space. In addition to the facilities shown here, specific areas of memory can be accessed in **DEFT Macro/6809** assembly language via the *Separate Compilation* facilities and the *Assembler Interface*. However, the facilities described in this section can be used entirely within Pascal and results in *position independent code (PIC)*.

4.1 BYTE and WORD Arrays

Absolute memory can be accessed as BYTEs or WORDs by using the corresponding pre-defined *array*. BYTE is ARRAY[$0000..$FFFF] OF 0..255 and WORD is ARRAY[$0000..$FFFF] OF INTEGER. The subscript used represents the actual memory address that is used. Example:

```
IF BYTE[1024] <> $41 THEN BYTE[1024] := $42;
WORD[$7FFE] := $FFFF
```

4.2 Absolute Address Operator (@)

The absolute integer address of any variable can obtained with the unary operator @. Example:

```
WORD[@I] := 5;
I := 5
```

The above two statements are equivalent. This facility can be combined with the *ptr* builtin function to put the address of any variable into a *pointer* type variable.

4.3 CALL Function

The predefined function CALL provides the ability to invoke the machine language functions and subroutines typically found in the Color Computer's ROM. The Function definition is:

```
TYPE ROMAddress = Integer;
     ARegister = 0..255;

FUNCTION CALL (RtnAddress : ROMAddress;
                    Parm : ARegister) : Aregister
```

When using the CALL function, the first parameter is the absolute memory address of the subroutine to be invoked. The second parameter is the value to be passed in the *A* register. The value

returned by the function is the value that the subroutine returned in the *A* register. Example:

```
REPEAT Key := CALL (WORD[$A000],0) UNTIL Key <> 0
```

The above example invokes the ROM subroutine whose address is located at absolute memory WORD $A000 (POLCAT). A zero is passed to this routine in the *A* register and the value returned by the subroutine is stored in the variable *Key*. The effect of the *repeat* statement is to wait until a keystroke is entered at the keyboard and to store the keystroke in *Key*. NOTE: In order to access ROM routines, you will have to run your program in 32K mode.

5 Static Variable Allocation

In the section *Variables* in the Pascal Language Summary, the standard automatic allocation scheme of Pascal is described. This is the default variable allocation incorporated into **DEFT Pascal**. However, it is also possible to *statically* allocate a variable.

When a variable is statically allocated, memory is reserved at compile time. This means that every time the variable is accessed, the *same* memory area is accessed *even if the block that the variable is defined in has been deactivated and then reactivated.*

This allows you to store a value into a statically allocated variable that is local to a procedure, before exiting from the procedure. Then when the procedure is subsequently invoked, be able to access that variable and retrieve the previously stored value. This can't be done with automatically allocated variables since the specific memory location occupied by the variable may change on each allocation.

5.1 STATIC Attribute

Variables are statically allocated when one of several *attributes* are added to the *var* statement in which they are defined. An attribute is a keyword which immediately follows the *var* keyword. The simplest of these attributes is the keyword *static*. The only result of this attribute is to cause all variables defined in the current *var* statement to be statically allocated. Example:

```
VAR          A : Char;
VAR STATIC   B, C : Integer;
             D : Char;
VAR          E, F : Integer;
             G : Char;
```

In the above example, variables *B*, *C* and *D* are all statically allocated. Variables *A*, *E*, *F* and *G* are all dynamically allocated. The scope of all the variables is the same.

5.2 PUBLIC Attribute

The *public* attribute, like the *static* attribute, causes all the variables defined in the corresponding *var* statement to be statically allocated. However, the *public* attribute can only be used in *var* statements at the *PROGRAM* or *MODULE* (see *Separate Compilation*) level and may not be used in *var* statements in *procedures* or *functions*.

In addition to causing a variable to be statically allocated, the *public* attribute *extends the scope* of the affected variables to other *separately compiled modules*. These other modules reference these *public* variables by declaring the same variables using the *external* attribute (see below). Example:

```
VAR PUBLIC      A, B : Char;
                C : Integer;
```

In the above example all three variables are statically allocated and made public. See The section on *Separate Compilation* for more information.

5.3 EXTERNAL Attribute

The *external* attribute is the complementary attribute to the *public* attribute. All variables defined in a *var* statement with the *external* attribute are *not actually allocated* by that *var* statement. This statement causes the static allocation performed by the *var public* statement to be used. Example:

```
VAR EXTERNAL    A, B : Char;
                C : Integer;
```

In the above example the variables *A*, *B* and *C* have been declared *public* in another module where memory for them has been allocated. All references to *A*, *B* and *C* in the module with the *external* attribute will access the publicly defined variables. See the section on *Separate Compilation* for more information.

6 Separate Compilation

This section details a facility in the **DEFT Pascal** Compiler that allows a programmer to break up a large program into a number of smaller programs. These smaller programs (known generically as *modules)* can then be compiled and (usually) tested independently. One of the primary advantages of separate compilation is the additional level of identifier scoping that is provided.

6.1 Rationale

In general, identifiers (constants, types, variables, procedures and functions) defined within a module are known only within that module. These identifiers are thought of as *private* and are not known to other modules. Of course if *all* the identifiers are private then there is no way for the module to be used. For this reason some identifiers are always made *public* so that *controlled* access to the module is assured.

For example, a complete set of routines to handle high-resolution graphics could be a module. Some of these routines would be called from outside the module and would consitute the *interface* to your graphics package. These routines would be declared *public.*

Other routines would be utilities whose express purpose is to perform functions common to several of the *public* routines. These utility routines would remain private so that they would not be inadvertantly invoked by other *modules.* This also ensures that their names would not conflict with other names used in other *modules.*

The variables used by this graphics module are also divided into public and private. The public variables may provide a means to pass data to or from several of the procedures in the module or may be used to specify operational modes. The private variables would be used to store temporary or intermediate results.

A special **DEFT Pascal** language construct, called an *interface module*, could be used to provide the compile time linkage between the graphics module and those other modules that use it. This *interface module* would be included at the beginning of the other modules and would provide all the *external* declarations for the *public* procedures, functions and variables. In addition, it would include const and type statements in order to define any special constants or types required by the graphics module.

6.2 MODULE Block

In standard Pascal, a complete program is a self-contained unit. For many smaller programs this is quite adequate and provides a simple environment in which to develop them. However, when you wish to divide your program into several relatively independent pieces; you have a problem if these pieces do not map, one-to-one, into procedures or functions. It is this problem that **DEFT Pascal's** *module* solve.

A *module* is a **DEFT Pascal** construct that allows you to group a set of procedures, functions and variables into a sort of a self-contained subprogram which is compiled by itself. This Pascal *module* can then be combined with other Pascal *modules*, **DEFT Macro/6809** Assembler *modules* and to only *one* Pascal *program*, via **DEFT Linker**, to create a complete program.

The syntax of a MODULE is as follows:

```
MODULE <module name>;
  CONST <identifier> = <constant>;
                .
                .
                .
  TYPE <identifier> = <type definition>;
                .
                .
                .
  VAR <identifier> : <type definition>;
                .
                .
                .
  PROCEDURE <identifier> <parameter definition>;
            <block>;
                .
                .
                .
  FUNCTION <identifier> <parameter definition>;
           <block>;
                .
                .
                .
END.
```

As you can see, this is *almost* the same as a *program*. In fact, with **DEFT Pascal**, a *program* is merely a special type of *module*. A *program* is the only *module* which contains its own BEGIN <executable statements> END. It is with these <executable statements> in the final binary program that execution begins.

One other difference between a *program* and a *module* is *variable allocation*. In a *program*, the default allocation is *automatic*. In a *module*, the default type of allocation is *static*. Since there is no way of *explicitly* specifying automatic allocation, a *module's* variable types are all static. The primary reason for this is that there is no *frame structure*, (see *Assembler Interface*), for a *module* in which to automatically allocate a variable.

Linkage between *modules* and the *program* is provided via the *public* and *external* attributes described below.

6.3 PUBLIC Procedures and Functions

Public procedures and functions are declared at the outer most block level of a *program* or *module*, and contain a *public* attribute immediately following the procedure or function statement. Procedures and functions which are nested within other procedures or functions may not have the *public* attribute. The syntax of a *public* procedure is as follows:

```
PROCEDURE <identifier> <formal parameter definition>;

  PUBLIC;

  <declaration statements>

  BEGIN
  <executable statements>
  END
```

The only difference between this and a standard procedure (or function) is the *public* attribute immediately following the *procedure* or *function* statement.

Once a procedure or function has been declared *public*, it may be invoked from other *modules* which have declared the *same* procedure or function as *external* (see *EXTERNAL Procedures and Functions*). Note: you may not use the same identifier to declare a procedure, function or variable as *public* in more than one *module*. However, once it is declared as *public*, you may declare it as *external* in as many *modules* (or the *program*) as you wish. An identifier cannot be declared as both *public* and *external* in the same *module* or *program*.

6.4 EXTERNAL Procedures and Functions

An *external* declaration allows a *public* procedure or function to be known and invoked in any *module* or *program* in which it is declared as *external*. A procedure or function is declared as *external* by following the procedure or function statement with only the *external* statement. The syntax is as follows:

```
PROCEDURE <identifier> <formal parameter definition>;
EXTERNAL
```

This type of procedure or function does not have a <block> associated with it. However, it *must* have a corresponding *public* procedure or function declared in another *module* whose procedure or function statement is *identical* to the one used with the *external* statement. Note that like the *public* statement, the *external* statement can be used only with procedures and functions which are declared at the outer most block level of a *module* or *program*.

6.5 PUBLIC and EXTERNAL Variables

The *public* and *external* attributes, in the VAR statement, cause static memory allocations to be made, as described in the section on *Static Variable Allocation*. *Public* variables (like *public* procedures) are those variables whose scope has been explicitly extended beyond the enclosing *module* or *program*. *External* variables are those variables which actually exist in other *modules* (OR the *program*) as *public* variables, but whose scope has been extended into this *module* or *program*.

As mentioned in the section on *public* procedures, you may not use the same identifier to declare a procedure, function or variable as *public* in more than one *module*. However, once it is declared as *public*, you may declare it as *external* in as many *modules* (or the *program*) as you wish. Any identifier cannot be declared as both *public* and *external* in the same *module* or *program*.

6.6 INTERFACE Block

An *interface* Block is a special **DEFT Pascal** Compiler construct which is used in conjunction with a *program* or *module*. Its purpose is to simplify the compile time *module* linkage (which would normally occur via *external* attributes and statements).

The *interface* block is an optional construct which may be included 1 or more times before the *module* or *program* statement. The syntax is as follows:

INTERFACE <interface name>;

<special declaration statements>

END

The <special declaration statements> are generally the same as <declaration statements> with the exception that all procedure, function and VAR statements are *assumed to be external*. That is,

procedure and function statements don't have *public*, FORWARD or *external* statements following them. Nor do they have <block>s following them. They are assumed to be *external*, since they are found in the *interface* Block.

VAR statements also cannot have *public*, *external* or STATIC attributes associated with them since they are assumed to be *external*.

6.7 Use Of INTERFACE Block

In general, you will create an *interface* block for each *module* that you create. The *module* will contain all the *public* definitions and will be compiled to create an object module that contains those procedures, functions and variables. The *interface* module will exist only in the form of Pascal source code and contain the *external* (by default) definitions that are then used in all the other *modules* (or the *program*) that reference this *module*.

In our graphics example, we might have the following *module*:

```
MODULE HiResolution;

CONST  ScreenSize = $1800;

TYPE   ScreenByte = -128..127; (* 1 Byte Integer *)
       Screen = ARRAY[1..ScreenSize] OF ScreenByte;
       GraphTypes = (GTalpa, GTsemi4, GTsemi6, ...);

VAR PUBLIC
       GraphMode : GraphTypes;

PROCEDURE MapScreen (VAR ScreenVar : Screen);
PUBLIC;
  .
  .    (* procedure block *)
  .

PROCEDURE ClearScreen (VAR ScreenVar : Screen);
PUBLIC;
  .
  .    (* procedure block *)
  .

  .
  .    (* other public and private procedures
  .    and functions required for package *)
  .

END.
```

This *module* contains a number of *public* interfaces including procedures, functions and at least one variable. Another *module* which is responsible for creating pie-charts may reference this *module* as follows:

```
INTERFACE HiResolution;

CONST  ScreenSize = $1800;

TYPE   ScreenByte = -128..127; (* 1 Byte Integer *)
       Screen = ARRAY[1..ScreenSize] OF ScreenByte;
       GraphTypes = (GTalpa, GTsemi4, GTsemi6, ...);

VAR    GraphMode: GraphTypes;

PROCEDURE MapScreen (VAR ScreenVar : Screen);

PROCEDURE ClearScreen (VAR ScreenVar : Screen);

.

END;

.

MODULE PieCharts;

.
.

END.
```

The module *PieCharts* uses the module *HiResolution* and sees its interface to *HiResolution* in terms of the *interface* block. Note that in general, the source code comprising the *interface* block will be in an independent file which is copied at compile time via the compiler *%C* directive.

One final note, the file *PASCALIB/EXT* is actually an *interface* block with a *%N* at the beginning and a *%L* at the end which is automatically copied by **DEFT Pascal** at the beginning of every compilation. You can force it to be listed by including an *L* directive in the directive prompt on the compiler startup screen.

7 Assembler Interface

One of the primary advantages to using both **DEFT Pascal** and **DEFT Bench** is the ability to *easily* mix Pascal and assembler language as appropriate in the development of a program. This section provides the information on using variables, procedures and functions from assembler and in turn creating variables, procedures and functions in assembler for use from Pascal, with **DEFT Pascal.**

A pre-requisite required for this section is a familiarity with the Motorola 6809 Assembler Language, and the **DEFT Macro/6809** Assembler. Information on linking object files produced by the **DEFT Pascal** Compiler and the **DEFT Macro/6809** Assembler can be found in the section on the **DEFT Linker.**

7.1 Code Generation Strategy

The **DEFT Pascal** compiler is a single-pass, recursive descent compiler which directly produces 6809 object code suitable for linking by **DEFT Linker.** In order to produce this object code, a *code generation strategy* is required so that the state of the machine can be predicted from statement to statement. This strategy defines how code, data and stack memory areas are organized as well as how the 6809 registers are used. In addition, the actual memory organization of all the various Pascal types should be understood.

Variable Sizes and the Stack

As can be guessed by the ordinal and pointer *types* available with **DEFT Pascal,** the language is 16 bit oriented. To a large extent this is due to the registers and functions available on the 6809. By keeping to a 16 bit organization, the resulting compiler is both smaller and more efficient.

In general, all instructions generated by the compiler are oriented around the program stack. As factors are encountered in an expression, they are pushed on the stack. Operators then operate on the top of the stack or combine the top two elements of the stack to form a result which is left on the top of the stack.

The number of bytes of data pushed on the stack depends on the *type* of the expression. The following table shows the number of bytes for each *type:*

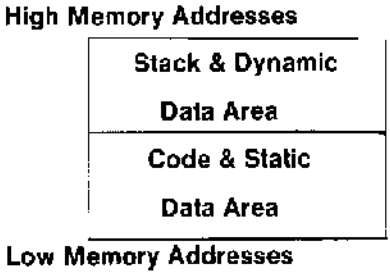
ordinal type	**2 bytes**
pointer type	**2 bytes**
real type	**7 bytes***
set type	**32 bytes**
file type	**286 bytes + type size**
string type	**string size + 1 bytes**
array and record types	**sum of components**

Although real types have a size of 6 bytes, when a real type is pushed on the stack, an additional byte is added in order to limit loss of precision during arithmetic operations. The symbol table printed at the end of each block shows the size of all the variables and types defined within that block.

Any time parameters are passed to a procedure or function, they are first pushed onto the stack. Values returned by functions are left on the stack when the function returns.

Memory Organization

The general memory organization of a Pascal program is shown in the following diagram:



As can be seen from the above diagram, the code and static data items are allocated in low memory and the stack with its associated dynamic data items are allocated in high memory.

The code and static data items are interspersed in the order in which they were encountered by the compiler. The code and static data area is built from low addresses to high addresses by the compiler. The resulting area is what is linked by **DEFT Linker** and eventually loaded via the *LOADM* command. Because **DEFT Linker** essentially handles all the code and static data linkage, the actual organization of memory is of little concern to the programmer.

The stack and dynamic data area is organized by the compiler but not actually allocated until execution of the resulting program. As a result the actual memory addresses cannot be predicted. The organization of this stack area is the key to interfacing Pascal and Assembler.

Register Usage

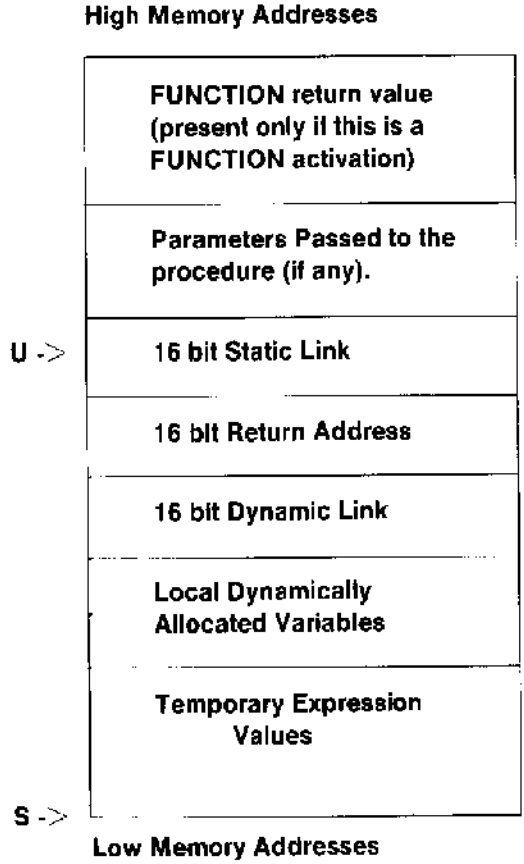
The use of the registers is oriented around the stack. The following lists the 6809 registers and summarizes their use:

- The *S register* is the program stack register. It always points to the youngest element on the stack. This stack always grows or shrinks by the size of the *type* begin pushed or popped. Elements are added by decrementing the S register and are removed by incrementing the S register.
- The *D register* is the primary accumlator, and so is considered to be the *top of stack* for most operations. This is done by placing data in the *D* register before actually pushing it on the stack. Data is popped from the stack into the *D* register. By considering the *D* register to be the *top of stack*, operating on the top stack element is easy with the 6809 instruction set.
- The *U register* is the *frame pointer*, which identifies that group of data on the stack which is associated with the most recent procedure activation. See the section on *Procedure Frame Structure* for a complete description.
- The *X register* is used as a secondary *frame pointer*, when traversing the static frame links in order to access an identifier which is *global* to a procedure. See the section on *Procedure Frame Structure* for a complete description. It is also used for array indexing and variable addressing.
- The *Y register* is used for temporary storage, loop counting and compare operations.

On return from a procedure or function, only the *U*, *S* and *DP* registers will be preserved. All other registers may have been modified.

7.2 Procedure Frame Structure

A *frame* is a contiguous portion of the stack that contains all the dynamic information relating to a specific procedure activation. Anytime a procedure or function is invoked, a frame is pushed onto the stack. The structure of a frame is:



The *base* of the frame is the *static link*. The U register always contains the base address of the most recently active frame (last one pushed on the stack). The following notes apply to the individual fields of the frame:

1. The function return value is only present on a function activation and can be considered to be the "zeroth" parameter.

2. The parameters are pushed on the stack in the order in which they occur in the <parameter list>. That is, the first parameter has the highest memory address and the last parameter has the lowest memory address. Each occupies the amount of memory specified in the section on *Variable Memory Requirements.*

3. The *static link* contains the base address of the most recent frame activation for the immediately enclosing procedure. This address is used when referencing variables which are global to the current procedure.

4. The 16 bit return address is the last element of the frame that is created by the *calling procedure* with a JSR or BSR instruction. The *called procedure* creates the remainder of the frame before executing its first statement.

5. The 16 bit *dynamic link* is the base address of the *calling procedure's* frame. It is placed on the stack by the *called procedure* via a PSHS U instruction. The U register is then immediately reset to the current frame's base address via a LEAU 4,S instruction.

6. The local, dynamically allocated variables are then allocated via an LEAS -n,s instruction which *only allocates and does not initialize.*

7. As <executable statement>s are executed additional stack space is used for temporary, intermediate expression values.

Returning from a procedure is easily accomplished with the following two instructions: LEAS -4,U and PULS U,PC. The *calling procedure* is then responsible for removing the parameters from the stack and using the function return value (if there is one).

The reason for having separate static and dynamic links is to provide for the ability to handle recursive procedure (or function) activation. The static link provides execution time identifier scoping, regardless of the number of times the current procedure has activated itself. The dynamic link provides the ability to return to the frame that activated the current procedure (or function).

As can be seen, as long as the assembly language program obeys these rules, it can either invoke a Pascal procedure or function or be invoked as if the assembly language procedure or function was written in Pascal.

7.3 Linkage

Linkage between Pascal *modules* is implemented via *public* and *external* attributes and statements as described in previous sections. Linkage to assembly language modules is exactly the same.

You can declare your own Pascal callable routine as *public* in your assembly language program so that it is visible to **DEFT Linker**. You then use the same name to declare the corresponding *external* procedure or function in the Pascal *module(s)* from which it is to be called. The same is true of shared, static variables which would be declared as *public* in your assembly language modules and *external* in the appropriate Pascal *module*.

Alternatively, you can create a Pascal *interface* that corresponds to your assembly language module in order to provide a more formal interface. All Pascal *modules* that reference any of your assembly language procedures, functions or variables would then %C the *interface module* to the beginning of their code. Language identifiers that are declared *external* in Pascal must be declared as *public* in your assembly language program.

Any Pascal procedures, functions or variables you wish to access from assembly language, must be declared as *public* in the corresponding Pascal *module* or *program*. The identifiers are then declared as *external* (via the EXT directive) in your assembly language program.

7.4 Initialization

All programs produced by **DEFT Linker** have a first instruction. For Pascal programs produced with the **DEFT Pascal** Compiler, this is in the runtime support module named PASBOOT. This is the module that determines the amount of memory in your system, sets the stack pointer appropriately, sets up all interrupt vectors for the device drivers, setups the initial frame on the stack and then calls the main pascal program.

7.5 PIC and ROM

The code produced by the **DEFT Pascal** Compiler is generally position independent and non-selfmodifying (can be placed in Read Only Memory-ROM). There are certain conditions under which this is not true:

1. Any presence of *static* or *public* variables within a Pascal program will result in a module that is self-modifying.

2. Any procedure, function or variable that is declared as *external* in a Pascal program, and whose actual address is an absolute memory location, will result in a module that is not position independent. Absolute memory access can be accomplished in **DEFT Pascal** via the BYTE, WORD and CALL language elements so that the resulting module *will* be position-independent.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X