DEFT Extra

User's Guide

Tandy Color Computer Software Series

Version 1 First Printing

DEFT Extra User's Guide

DEFT Extra

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 Extra

1 Introduction

DEFT EXTRA is a set of graphics, gaming and direct I/O routines for use with the **DEFT Pascal Workbench, DEFT Pascal** or **DEFT Bench.** This **DEFT EXTRA User's Guide** describes how to use these routines from Pascal and also contains a technical reference section which describes the assembler interfaces to the **DEFT Pascal** runtime library as well as the format of the **DEFT** object files.

1.1 Diskette Contents

The following files are contained on the distribution diskette:

1. *EXTRA/EXT* - this is a Pascal INTERFACE file which contains all the necessary declarations for using the library.
2. *EXTRA/LIB* - this is an object library which contains all the actual routines. The members of the library are:
   - XGDRAW - contains GDRAW.
   - XGPAINT - contains GPAINT.
   - XGCIRCLE - contains GCIRCLE.
   - XGBOX - contains GBOX
   - XGLINE - contains GLINE
   - XGMAIN - contains GINIT, GDISPLAY, GDISPLAYTEXT, GCLS, GSET and GPOINT
   - XGSUPPRT - contains support routines and tables for use by the preceding graphics modules.
   - XJOY - contains JOYSTICK and JOYFIRE.
   - XSOUND - contains SOUND.
   - XSNDJOY - contains support routines and tables for use by XJOY and XSOUND.
   - XDIRECT - contains OPENDIRECT, POSITIONFILE, UPDATEFILE and APPENDFILE
   - XDIRECTP - contains support routines for XDIRECT.
3. *PAINT/PAS* - this is an example Pascal program that uses the routines in **DEFT EXTRA** to create a crude painting program.

4. *PAINT/OBJ* - this is the object form of PAINT/PAS.
5. *PAINT/BIN* - this is the final binary form of PAINT/PAS linked with EXTRA/LIB.
6. *DISKREC/PAS* - this is an example Pascal program that uses the routines in **DEFT EXTRA** to create a simple file maintenance program.
7. *DISKREC/OBJ* - this is the object form of DISKREC/PAS.
8. *DISKREC/BIN* - this is the final binary form of DISKREC/PAS linked with EXTRA/LIB.

1.2 Using DEFT EXTRA

To use **DEFT EXTRA** you have to do three things:

1. Copy the files EXTRA/EXT and EXTRA/LIB onto your work diskette.
2. Put a **%C EXTRA/EXT** directive at the beginning of your Pascal source program. This will cause the compiler to read in the INTERFACE file which defines all the constants, types, procedures and functions defined in **DEFT EXTRA**. If you want to get a listing of EXTRA/EXT, put a **%L** directive before the %C directive.
3. When you link your program, specify **EXTRA/LIB** as the *last* object file to be linked. This will cause the linker to include all the necessary modules from this library into your final binary program.

1.3 Example PAINT Program

This program uses **DEFT EXTRA** to provide a simple paint facility. After starting the program you will see a blue background with a menu on the left hand side. There will also be single blinking pixel on the screen. You can use the right joystick to move this single pixel cursor about.

To draw something you must select a draw operation and a color to draw with. You select an operation by positioning the cursor within the appropriate box and hitting the fire button. You will hear a beep to indicate that the operation was selected. You select a color by positioning the cursor over the appropriate color bar at the lower

left and hitting the fire button. You can select colors and operations independently. The operations are:

1. Paint in an area with the currently selected color using a border of the currently selected color. You can change the paint color by selecting a new color *after* selecting the paint operation. The border color will remain the same. You can specify the border color by selecting it before selecting the paint operation. To paint, position the cursor to where you want to start painting and hit the fire button.

2. Draw an empty box. Position the cursor at one corner and hit the fire button. Position it to the opposite corner and hit the fire button a second time. The box is then drawn.

3. Draw a full box. Same as draw empty box.

4. Draw a circle. Position the cursor to the center of the circle and hit the fire button. Position it to any point on the circumference and hit it again. The circle is then drawn. The delay that you notice is due to the PAINT program computing the radius of the circle.

5. Draw a line. Position the cursor to a point on the screen and hit the fire button. Position it again and hit the fire button again. The line is drawn and the beginning point of a second line is selected as the ending point of this last line.

6. Drag a line. Position the cursor to a point on the screen and hit the fire button. A single point is set. Move the cursor and a line will automatically be drawn as you move the cursor. Hit the fire button again to stop the operation.

To exit from the program, position the cursor to the extreme upper left-hand corner of the screen and hit the fire button.

2 Graphics

The routines in this section provide the same kinds of graphics capabilities that are found in Extended Basic. These routines allow you to easily draw lines, squares, boxes, circles, ellipses, arcs and odd shaped objects which you can then paint in. These operations as well as point set and interrogate are available in all 8 graphics modes.

2.1 Types and Constants

Before describing each routine, it is necessary to define a number of types and constants that are used by each. The INTERFACE file EXTRA/EXT contains all the definitions of these types and constants.

GMode - This is a type which represents one of the eight graphics modes:

```
TYPE GMode = (G1C, G1R, G2C, G2R, G3C, G3R, G6C, G6R);
```

A complete description of each of these modes can be found in *Getting Started with Color Basic*. No matter which mode that you use, you will be working in a coordinate system that contains 256 horizontal (X) points and 192 vertical (Y) points.

GColor - This is a type which represents one of the eight graphics colors:

```
TYPE    GColor =  (GGreen, GYellow, GBlue, GRed,
                   GBuff, GCyan, GMagenta, GOrange);

CONST   GHiResBlack = GGreen;
        GHiResGreen = GYellow;
        GHiResBuff = GHiResGreen;
```

The first four colors are the primary colors available in the graphics modes that end in **C**. The second four are the alternate colors for those graphics modes. The **GHiRes** colors are the colors available in the graphics modes that end in **R**. GHiResGreen is the primary color and GHiResBuff is the alternate color.

GData - This is a type which represents the standard parameters of a particular graphics mode:

```
TYPE  GData = RECORD
                RowShift    : 2..3;
                PageCount   : 2..12;
                XDivisor    : 1..4;
                YDivisor    : 1..3;
                YShiftCount : 6..8;
                XResolution : 64..256;
                YResolution : 64..192;
                END;
      GDataPtr = ^ GData;
```

where:

- **RowShift** is the log2 value of the number of bytes in a row.
- **PageCount** is the number of 512 byte pages being used.
- **XDivisor** is the amount that the X value should be divided by. It is equal to 256 DIV **XResolution**.
- **YDivisor** is the amount that the Y value should be divided by. It is equal to 192 DIV **YResolution**.
- **YShiftCount** is the log2 value of the **XResolution**.
- **XResolution** is the actual horizontal resolution.
- **YResolution** is the actual vertical resolution.

This information is used within the graphics routines and will usually not be needed by the programmer.

GBlock - This is a type which represents a graphics control block:

```
TYPE  GBlock = RECORD
                 Mode      : GMode;
                 Address   : Integer;
                 AltColor  : Boolean;
                 Table     : GDataPtr;
                 Draw      : RECORD
                   X, Y : Integer;
                   Color : GColor;
                   Angle : 0..7;
                   Scale : 1..127;
                   END;
                 END;
```

Where:

- **Mode** is the initialized graphics mode.
- **Address** is the memory address of the beginning of the graphics data.
- **AltColor** indicates whether the alternate color set is to be used. (TRUE => use alternate)
- **Table** is a pointer to the set of **GData** for this **Mode**.
- **Draw** is a record containing the current DRAW information including current X/Y coordinate, current color, angle adjustment and scaling factor. See **GDraw** for more information.

You will have to declare a variable of type **GBlock** for each graphics screen that you will be using.

2.2 GInit

This routine initializes a variable of type **GBlock**. This is the first routine that you should call. The declaration for **GInit** is:

```
PROCEDURE GInit (VAR GraphBlock : GBlock;
                 GraphMode : GMode;
                 GraphPage : Integer;
                 AltColor : Boolean;
                 Background : GColor;
                 DrawColor : GColor);
```

where:

- **GraphBlock** is your variable of type GBlock which is to be initialized.
- **GraphMode** is the graphics mode that you want to use for this particular graphics area.
- **GraphPage** is a number from 0 to 127 which specifies which 512 byte memory page the graphics area is to start on. The actual memory address used will be equal to this number multiplied by 512. It is important that you select a page which does not overwrite your program or the stack. There are basicly three places that the graphics area can go:

1. Below your program. In this case, you specify an origin to the linker which places your program above the area to be used by the graphics area.

2. Between your program and the stack. In this case, you look at the map produced by the linker to determine the uppermost byte of your program and the stack requirements of your program and then select an area between them that won't overwrite the top of your program, the heap or the bottom of the stack.

   NOTE: If you are going to use **GPaint**, you should not position your graphics page here since **GPaint** uses heap memory for its operation.

3. Above the stack. In this case, you must modify PASBOOT/ASM to start the stack lower in memory and then you can use the memory above this for graphics.

- **AltColor** indicates whether you want to use the primary or alternate color set. TRUE indicates to use the alternate color set.
- **BackGround** is the initial color that you want the graphics area to be set to. **GInit** automatically does a **GCls**.
- **DrawColor** is the initial color that you want to use for **GDraw**.

After calling **GInit** the specified graphics page will be initialized to the **BackGround** color and the values in the **Draw** sub-record of **GraphBlock** will contain the following values:

- **X, Y** will be set to 128,96.
- **Color** will be set to **DrawColor**.
- **Angle** will be set to 0.
- **Scale** will be set to 4.

For example:

```
%C EXTRA/EXT
PROGRAM TextExtra (Input, Output);
VAR Block : GBlock;
  ...
BEGIN
  GInit (Block, G6C, 10, False, GBlue, GYellow);
  ...
```

In this example, **Block** is the local variable that contains the control information for the graphics routines. We are going to use the **G6C** graphics mode which gives us 4 colors and uses 6K of memory. The graphics memory starts at memory location 5120 (10 * 512). We are going to use the primary color set, set the background initially to blue and start drawing with the color yellow.

2.3 GDisplay

This routine causes the specified graphics area to be displayed on the screen. You can have more than one graphics area that your program is working with. However, only 1 at time can be displayed. The declaration is:

```
PROCEDURE GDisplay (VAR GraphBlock : GBlock);
```

For example:

```
GDisplay (Block);
```

2.4 GDisplayText

This routine causes the standard text screen to be displayed. If you do any WRITEs or WRITELNs to the screen after doing a **GDisplay**, your text will be stored in the text screen area, however, it will not be displayed until you call **GDisplayText**. The declaration is:

```
PROCEDURE GDisplayText;
```

For example:

```
GDisplay (Block);
 ...
GDisplayText;
WRITELN ('ENTER X,Y: ');
READLN (X, Y);
GDisplay (Block);
 ...
```

In this example, the first **GDisplay** displays the graphics data. Later in the program when you want to prompt for some information, you use **GDisplayText** to setup the text screen so that you can. After getting the information, **GDisplay** sets the screen back to the graphics mode.

2.5 GCls

This routine causes the graphics area to be cleared to the specified **BackGround** color. The declaration is:

```
PROCEDURE GCls (VAR GraphBlock : GBlock;
                BackGround : GColor);
```

For example:

```
GCls (Block, GRed);
```

2.6 GSet

This routine causes a single point to be set. **X** and **Y** specify the coordinates and **PointColor** specifies what color it should be set to. The declaration is:

```
PROCEDURE GSet (VAR GraphBlock : GBlock;
                X, Y : Integer;
                PointColor : GColor);
```

For example:

```
GSet (Block, 187, 43, GYellow);
```

NOTE: if **X,Y** is outside the range (0..255,0..191) then no pixel is set.

2.7 GPoint

This routine returns the color of the specified point. X and Y specify the coordinates to interrogate. The declaration is:

```
FUNCTION GPoint  (VAR GraphBlock : GBlock;
                  X, Y : Integer) : GColor;
```

For example:

```
IF GPoint (Block, 187, 43) = GYellow THEN ...
```

NOTE: if X,Y is outside the range (0..255,0..191) then the returned value is unpredictable.

2.8 GLine

This routine causes a straight line to be drawn. X1, Y1 are the coordinates of one endpoint and X2, Y2 are the coordinates of the other endpoint. **LineColor** is the color to be used for the line.

```
PROCEDURE GLine (VAR GraphBlock : GBlock;
                 X1, Y1, X2, Y2 : Integer;
                 LineColor : GColor);
```

For example:

```
GLine (Block, X, Y, X+15, Y-15, GGreen);
```

NOTE: if either endpoint is outside of the range (0..255,0..191) the line will not be drawn.

2.9 GBox

This routine causes a box to be drawn. X1, Y1 are the coordinates of one corner of the box and X2, Y2 are the coordinates of the opposite corner. **BoxColor** is the color to use. **Solid** indicates whether the box should be filled in. TRUE indicates fill it in, FALSE indicates don't fill it in. The declaration is:

```
PROCEDURE GBox(VAR GraphBlock : GBlock;
               X1, Y1, X2, Y2 : Integer;
               BoxColor : GColor;
               Solid : Boolean);
```

For example:

```
GBox (Block, X, Y, X+15, Y-15, GGreen, True);
```

NOTE: if either **X1,Y1** or **X2,Y2** are outside the range (0..255,0..191) then the corresponding lines of the box are not drawn.

2.10 GCircle

This routine causes an arc to be drawn. The declaration is:

```
PROCEDURE GCircle (VAR GraphBlock : GBlock;
                   X, Y : Integer;
                   Radius : Integer;
                   Color : GColor;
                   Ratio : Integer;
                   Start, Finish : Integer);
```

where:

- **X, Y** are the coordinates of the center of an ellipse of which the arc is a part.
- **Radius** is the horizontal radius of the ellipse.
- **Color** is the color to use when drawing the arc.
- **Ratio** is the height to width ratio relative to 256. The vertical radius of the ellipse is equal to RADIUS * RATIO DIV 256. Using a ratio of 256 results in an equal radius all the way around. However, due to the pixel layout on the screen, this results in a slight vertical exaggeration. A Ratio of 224 more closely approximates a circle.
- **Start** and **Finish** indicate what portions of the arc that you want drawn. They are numbers in the range 0 to 63 where zero represents the 3 o'clock position, 16 the 6 o'clock position, 32 the 9 o'clock position and 48 the 12 o'clock position.

  You can specify **Start** greater than **Finish** to draw an arc through the 3 o'clock position. A full ellipse is indicated by specifying **Start** and **Finish** to be the same number.

For example:

```
GCircle (Block, 128, 96, 64, GRed, 224, 0, 0);
GCircle (Block, 255, 96, 230, GBlue, 96, 16, 48);
```

The first line draws a red circle in the middle of the screen. The second line draws the left half of an ellipse that is flattened on the top and bottom. This half of an ellipse takes up most of the screen.

NOTE: those portions of the specified arc that lie outside the range of (0..255,0..191) are not drawn.

2.11 GPaint

This routine causes an area of the screen bordered by one color to be filled in with (possibly) another color. **X, Y** are the coordinates of the point at which painting is to begin. **PaintColor** is the color to use while painting. **BorderColor** is the color at which to stop painting.

GPaint uses the heap to allocate small control blocks which indicate where all the outstanding places are to a painted. If there is insufficient memory to complete the operation, then GPaint will return a FALSE. Each control block is 7 bytes long and all are released from the heap when GPaint returns.

```
FUNCTION GPaint  (VAR GraphBlock : GBlock;
                  X, Y : Integer;
                  PaintColor : GColor;
                  BorderColor : GColor) : Boolean;
```

For example:

```
IF NOT GPaint (Block, 15, 80, GRed, GYellow)
THEN ... (* error *)
```

2.12 GDraw

This routine draws lines based on directions contained in an ASCII string. **Directions** is the ASCII string. Since **GDraw** does not modify **Directions**, you can use a constant when calling GDraw. The declaration is:

```
PROCEDURE GDraw(VAR GraphBlock : GBlock;
                VAR Directions : String);
```

The directions string can contain the following characters:

1. A leading **B** which indicates that the immediately following operation is not to result in a line being drawn. If this prefix is not present, then the following movement operation will result in a line being drawn from the current draw position to the new draw position.

2. A leading **N** which indicates that the immediately following operation is not to result in a change in the draw position. This allows you to draw a line and return to the original draw

position. If the **N** prefix is not present, then the end of the specified line becomes the new draw position.

3. An **M** (for move) which indicates that a new X,Y coordinate is to be specified. The number following the **M** is the new **X** coordinate. If a comma follows, then the next number is the new **Y** coordinate. If either number has a leading plus (+) or minus (-) then the corresponding value is relative to the old value. The initial X,Y coordinate position is 128,96.

4. A **U** (up), **D** (down), **L** (left), **R** (right), **E** (upper right), **F** (lower right), **G** (lower left) or **H** (upper left) which indicates a relative move in the specified direction. If a number follows, then it specifies the length of the move. If it is not present, then 1 is assumed.

5. An **A** which indicates that relative moves are to have their direction biased (clockwise) by some multiple of 45 degrees. The number following indicates the bias. If no number is present, then 0 is assumed. The initial default value is 0.

6. An **S** which indicates that relative moves are to be scaled by a specified factor. The number following is divided by 4 and the result is used to scale the amount of movement. The initial default value is 4.

7. A **C** which specifies a particular color to use in following operations. The number following the C is the ordinal value of the type **GColor** to use. The initial value is specified in the **GInit** call.

```
GDraw (Block, 'BM45,30;F3U5C2M+10A1URDLA');
```

The semi-colon (or any illegal character) can be used to improve readability.

3.1 JoyStick

This routine returns one of the coordinate values of one of the joysticks. The declaration is:

```
FUNCTION JoyStick (Select : Integer) : Integer;
```

The value returned is in the range 0..63. The values for **Select** are:

- **0** - right joystick horizontal coordinate.
- **1** - right joystick vertical coordinate.
- **2** - left joystick horizontal coordinate.
- **3** - left joystick vertical coordinate.

The value of the specified coordinate of the specified joystick is returned. For example:

```
X :– JoyStick (0) * 4;
Y := JoyStick (1) * 3;
```

This allows you to get X values in the range 0..252 and Y values in the range 0..189.

3.2 JoyFire

This routine returns an indication of whether the fire button of the specified joystick is being depressed. The declaration is:

```
FUNCTION JoyFire (Select : Integer) : Boolean;
```

The possible values for **Select** are:

- **0** - fire button on right joystick.
- **1** - fire button on left joystick.

A **True** returned value indicates that the button is being depressed. For example:

```
IF JoyFire (0) THEN ... (* do stuff *)
```

3.3 IRandom

This routine returns a pseudo-random integer equally distributed in the range 0..32767. The declaration is:

```
FUNCTION IRandom (VAR Seed : Integer) : Integer;
```

The number returned is dependent on the value of **Seed**. Each time that **IRandom** is called, **Seed** is updated to reflect the value returned. For example:

```
Die := SUCC (IRandom (Seed) MOD 6);
```

3.4 Random

This routine returns a pseudo-random real number equally distributed in the range 0.0 through 1.0. The declaration is:

```
FUNCTION Random (VAR Seed : Integer) : Real;
```

Random functions by invoking **IRandom** and dividing the result by 32767. For example:

```
Sample := Random (Seed) * Population;
```

3.5 Sound

This routine generates a sound of a specified **Frequency, Duration** and **Volume.** The declaration is:

```
PROCEDURE Sound (Frequency, Duration, Volume : Integer);
```

where:

- **Frequency** is a number in the range 1..255 specifying the frequency of the sound. A number 9 less than that given to Color Basic will produce about the same pitch.
- **Duration** is the length of time in 16ths of a second that you want to make the sound persist.
- **Volume** is a number in the range 0..31 specifying the volume of the sound. Zero indicates silence, 31 indicates full volume.

For example:

```
Sound (150, 16, 24);
```

4 Direct File I/O

The routines in this section provide the capability of directly accessing records in a disk file. These routines allow you to position to a particular record by record number and then read or update the record. In addition, you can add records to the end of an existing file.

4.1 Types and Constants

Before describing the routines, it is necessary to define the types that are used by each. The INTERFACE file EXTRA/EXT contains all the definitions of these types.

SectorData - This is a type which represents one sector's worth of data:

```
TYPE SectorData = ARRAY[0..255] OF Char;
```

DirectData - This is a type which represents the information used by the direct I/O routines:

```
TYPE DirectData = RECORD
                    FirstGranule : Integer;
                    GranuleTable : SectorData;
                    END;
```

4.2 OpenDirect

This routine initializes a variable of type **DirectData** for later use by **PositionFile** and **AppendFile**. The declaration is:

```
FUNCTION OpenDirect  (VAR Disk : FILE OF ...;
                      VAR Table : DirectData) : Boolean;
```

The file **Disk** must be a *typed* file (not **Text** or **FILE OF Char**) and must have been openned via a **RESET**. **Table** is a variable of type **DirectData** that **OpenDirect** is going to initialize. For example:

```
RESET (DiskFile, FileName);
IF NOT OpenDirect (DiskFile, Table)
THEN ... (* error *)
```

OpenDirect returns a TRUE if the open was successful. If it returns a FALSE, you can check the error status of the specified file (via **FILEERROR**) to determine the cause.

4.3 PositionFile

This routine positions the file to a particular record number within the file. The declaration is:

```
FUNCTION PositionFile    (VAR Disk : FILE OF ...;
                          VAR Table : DirectData;
                          RecNumber : Integer) : Boolean;
```

Where **Table** is the variable that was previously initialized by **OpenDirect** and **RecNumber** is an integer in the range 0..32767. The first record in the file is numbered 0.

After you position the file, you can either use **Get** to read the specified record or **UpdateFile** to write out the current record. For example to position and then read a record into DiskRec:

```
IF PositionFile (DiskFile, Table, RecNumber)
THEN BEGIN
  Get (DiskFile);
  DiskRec := DiskFile^;
  ...
```

PositionFile returns a TRUE if the operation was successful. If it returns a FALSE, you can check the error status of the specified file (via FILEERROR) to determine the cause. If FILEERROR returns a zero, then the cause was a record number that was outside the limits of the file. In this case, the position of the file is not changed.

4.4 UpdateFile

This routine writes the record in the file window to the current file position and then positions the file to the next record. The declaration is:

```
FUNCTION UpdateFile (VAR Disk : FILE OF ...) : Boolean;
```

An example update sequence would look like this:

```
IF PositionFile (DiskFile, Table, RecNumber)
THEN BEGIN
  Get (DiskFile);
  DiskRec := DiskFile^;
    ... (* modify DiskRec *)
  IF PositionFile (DiskFile, Table, RecNumber)
  THEN BEGIN
    DiskFile^ := DiskRec;
    IF NOT UpdateFile (DiskFile) THEN ...
```

UpdateFile immediately writes the record to disk and returns a TRUE if the update was successful. If it returns a FALSE, you can check the error status of the specified file (via FILEERROR) to determine the cause.

4.5 AppendFile

This routine adds the record currently in the file window to the end of a file which is being used for direct access. The declaration is:

```
FUNCTION AppendFile  (VAR Disk : FILE OF ...;
                      VAR Table : DirectData) : Boolean;
```

Although you can use this routine to sequentially build a file, it is not designed for that purpose. Its purpose is to add records to an existing file which is being updated directly. It is a relatively expensive routine in that it forces the record to be written to disk immediately and the disk directory to be updated. Use the regular **Rewrite, Write** and **Close** I/O statements to efficiently build a file from scratch. An example of use:

```
DiskFile^ := Data;
IF NOT AppendFile (DiskFile, Table)
THEN ... (* error *)
```

AppendFile returns a TRUE if the append was successful. If it returns a FALSE, you can check the error status of the specified file (via FILEERROR) to determine the cause.

4.6 DeleteFile

This routine deletes an openned file from the diskette. The file can have been openned via either a REWRITE or RESET. The declaration is:

```
PROCEDURE DeleteFile (VAR DiskFile : Text);
```

For example:

```
RESET (DiskFile, FileName);
DeleteFile (DiskFile);
```

4.7 Overall Example

The following example is a very simple file maintenance program that uses DEFT EXTRA to maintain a direct access file.

```
%C EXTRA/EXT
PROGRAM DiskDirect (Input, Output);

TYPE   DiskData = RECORD
                    Data:STRING(20);
                    END;

VAR    I : Integer;
       FileName : String;
       DiskRec : DiskData;
       DiskFile : FILE OF DiskData;
       Table : DirectData;
       Command : Char;

BEGIN
  PAGE;
  WHILE TRUE DO BEGIN
    WRITE ('ENTER COMMAND: ');
    READLN (Command);
    CASE Command OF
     'X' : EXIT; (* exit from program *)
     'C' : BEGIN (* create a file *)
             WRITE ('CREATE FILE:');
             READLN (FileName);
             REWRITE (DiskFile,FileName);
             CLOSE (DiskFile);
             END;
     'O' : BEGIN (* open a file *)
             WRITE ('OPEN FILE:');
             READLN (FileName);
             RESET (DiskFile,FileName);
             WRITELN ('OPEN: ',
                OpenDirect (DiskFile, Table));
             END;
     'S' : BEGIN (* sequencially read a file *)
```

```
            WHILE NOT EOF (DiskFile) DO BEGIN
              READ (DiskFile, DiskRec);
              WRITELN (DiskRec.Data);
              END;
            END;
      'A' : BEGIN (* append a record *)
            WRITE ('DATA: ');
            READLN (DiskRec.Data);
            DiskFile ^ := DiskRec;
            WRITELN ('APP: ',
               AppendFile (DiskFile, Table));
            END;
      'R' : BEGIN (* read a record *)
            WRITE ('RECORD NUMBER: ');
            READLN (I);
            WRITELN ('POS: ',
               PositionFile (DiskFile, Table, I));
            GET (DiskFile);
            DiskRec := DiskFile ^ ;
            WRITELN (DiskRec.Data);
            END;
      'U' : BEGIN (* update a record *)
            WRITE ('RECORD NUMBER: ');
            READLN (I);
            WRITE ('DATA: ');
            READLN (DiskRec.Data);
            DiskFile ^ := DiskRec;
            WRITELN ('POS: ',
               PositionFile (DiskFile, Table, I));
            WRITELN ('UPD: ', UpdateFile (DiskFile));
            END
      ELSE WRITELN ('ILLEGAL COMMAND')
      END;
    END;
  END.
```

This program prompts the user for a one character command. The command entered is used in the main CASE statement to determine what function to perform:

- X indicates to terminate the program.
- C indicates to create an empty file. Normally you would also WRITE out all the initial records in the file. In this program, we

will use the **A** command to put records in the file.

- **O** indicates to open an existing file for direct access. The result of the **OpenDirect** call (1 or 0) is displayed to indicate whether the open was successful.
- **S** indicates to read the file from the current position. The record currently in the file window is the first one printed out. If the last function was an **R** or **U**, then the record associated with that call will be displayed.
- **A** indicates to add a record to the file. The result of the **AppendFile** call is displayed on the screen. Notice that you put the record to be written in the file window before calling **AppendFile.**
- **R** indicates to read a particular record. The result of the **PositionFile** call is displayed. After positioning, a **Get** is used to actually read the data into the file window. An assignment statement is then used to copy the data into a variable.
- **U** indicates to update a particular record. The result of the **PositionFile** and **UpdateFile** calls are both displayed on the screen. Notice that you put the record in the file window before calling **UpdateFile.**

You don't use the **Close** statement on a file openned for direct access. This is because there is no buffering of data to be written to disk.

5 Absolute Sector I/O

These two routines provide the ability to read and write to absolute sectors on disk.

5.1 ReadSector

This routine reads in an absolute sector. The declaration is:

```
FUNCTION ReadSector  (Drive, Track, Sector : Integer;
                      VAR Data : SectorData) : Boolean;
```

where:

- **Drive** is the diskette drive number (0..3).
- **Track** is the track number (0..34).
- **Sector** is the sector number (1..18).
- **Data** is the returned data.

ReadSector returns a TRUE if the read was successful and a FALSE if an I/O error occurred.

5.2 WriteSector

This routine writes out an absolute sector. The declaration is:

```
FUNCTION WriteSector  (Drive, Track, Sector : Integer;
                       VAR Data : SectorData) : Boolean;
```

where:

- **Drive** is the diskette drive number (0..3).
- **Track** is the track number (0..34).
- **Sector** is the sector number (1..18).
- **Data** is the data to write out.

WriteSector returns a TRUE if the write was successful and a FALSE if an I/O error occurred.

6 Technical Information

This portion of the **DEFT EXTRA User's Guide** describes the major internal Pascal runtime library interfaces. You can use this information to directly call these routines from assembly language or to replace them with your own routines. In general, you can replace an individual routine by modifying PASBOOT to patch in a JMP to your routine at the beginning of the routine to be replaced.

You access these routines by executing an LBSR or JSR to the symbol name given below and then declaring that symbol in your assembler program with an EXT statement. You may first have to either load registers or push parameters on the stack. Values will be returned either in registers or on the stack. The symbol names and calling conventions are described for each routine. Unless otherwise stated, the values of the D, X, Y and CC registers will be modified by the routine.

6.1 I/O Routines

The routines in this section make up the bulk of the runtime library and are responsible for providing ASCII/Binary data conversion, buffering and data transfer to and from disk, cassette, keyboard, screen and printer.

File Control Block (FCB) Format

In order to use any of these routines, you will have to supply a File Control Block or FCB. The definitions of the fields in the FCB are as follows:

```
FCBPOINTER   EQU  0   File Pointer
FCBNAME      EQU  2   File Name
FCBEXT       EQU  10  File Name Extension
FCBDEVNO     EQU  12  Device Number
FCBSTATE     EQU  14  State of Last Operation
FCBTYPE      EQU  15  File Type
FCBGRAN      EQU  16  Current Granule
FCBNEXTG     EQU  17  Next Granule
FCBTRACK     EQU  18  Track Number
FCBSECTR     EQU  19  Current Sector
FCBLSECT     EQU  20  Last Sector in Granule
FCBOPEN      EQU  21  Open Type
FCBINDEX     EQU  22  Index into FCBBUFR
FCBBUFSZ     EQU  24  Current Buffer Size
FCBLAST      EQU  26  Last Sector Size
```

```
FCBTYPESIZE   EQU  28   Record Type Size
FCBBUFR       EQU  30   Sector Buffer
FCBBASESIZE   EQU  286  Base Size of FCB
FCBRECORD     EQU  286  Record Offset
```

The FCBOPEN field is initialized via either the DFTRESET or DFTREWRITE routines. Its possible values are:

```
FCBOPENREAD   EQU  $AA  Open for seq read
FCBOPENWRITE  EQU  $CC  Open for seq write
```

The FCBSTATE field is initialized via DFTRESET or DFTREWRITE and then updated by each of the other I/O routines. Its possible values are:

```
FCBSTOKAY     EQU  0    Successful Operation
FCBSTEOF      EQU  $FF  End of File
FCBSTIOERR    EQU  $FE  I/O Error
FCBSTNOTFND   EQU  $FD  File not found
FCBSTILLEGAL  EQU  $FC  Illegal Operation
FCBSTFULL     EQU  $FB  Device Full
```

In general, on return from any of the following routines you will have to check the FCBSTATE field to determine whether the corresponding operation completed successfully.

DFTRESET (in PASIO) - This routine prepares an FCB for sequential input. The calling conventions are as follows:

On Entry:

- X contains the FCB address.
- Y contains the address of a STRING containing a standard file name.
- D contains the address of a STRING containing the default file name extension.

On Return: FCBSTATE contains the result.

DFTREWRITE (in PASIO) - This routine prepares an FCB for sequential output. The calling conventions are as follows:

On Entry:

- X contains the FCB address.

- Y contains the address of a STRING containing a standard file name.
- D contains the address of a STRING containing the default file name extension.

On Return: FCBSTATE contains the result.

DFTCLOSE (in PASIO) - This is the same as the CLOSE routine except that the calling conventions are as follows:

On Entry: X contains the FCB address.

On Return: FCBSTATE contains the result.

DFTREADCHAR (in PASIO) - This routine is used to read a single character from the next line in a file. It is used by the READ and READLN Pascal constructs.

On Entry: Before calling this routine you must push three 16 bit parameters onto the stack in the following order:

1. FCB address
2. address at which to place the character
3. field size (ignored, but should be 1)

On Return: FCBSTATE contains the result and the last two parameters are popped from the stack. The FCB address will remain on the stack.

DFTREADINT (in PASIO) - This routine is used to read the ASCII equivalent of a single integer from the next line in a file and convert it into binary. It is used by the READ and READLN Pascal constructs.

On Entry: Before calling this routine you must push three 16 bit parameters onto the stack in the following order:

1. FCB address
2. address at which to place the binary integer
3. binary integer byte size (either 1 or 2)

On Return: FCBSTATE contains the result and the last two parameters are popped from the stack. The FCB address will remain on the stack.

DFTREADREAL (in PASREAL) - This routine is used to read the ASCII equivalent of a single real number from the next line in a file and convert it into binary. This routine is used by the READ and READLN Pascal constructs.

On Entry: Before calling this routine you must push two 16 bit parameters onto the stack in the following order:

1. FCB address

2. address at which to place the binary real number

On Return: FCBSTATE contains the result and the last parameter is popped from the stack. The FCB address will remain on the stack.

DFTREADSTRG (in PASIO) - This routine is used to read the next line in a file into a STRING variable. It is used by the READ and READLN Pascal constructs.

On Entry: Before calling this routine you must push three 16 bit parameters onto the stack in the following order:

1. FCB address

2. address at which to place the STRING

3. maximum STRING size

On Return: FCBSTATE contains the result and the last two parameters are popped from the stack. The FCB address will remain on the stack.

DFTREADTYPE (in PASIO) - This routine is used to read a number of bytes from a file. This routine is used by the READ Pascal construct.

On Entry: Before calling this routine you must push three 16 bit parameters onto the stack in the following order:

1. FCB address

2. address at which to place the bytes read

3. the number of bytes to read

On Return: FCBSTATE contains the result and the last two parameters are popped from the stack. The FCB address will remain on the stack.

DFTREADLN (in PASIO) - This routine is used to read past the next carriage return in a file. It is used by the READLN Pascal construct in order to skip past the end of line.

On Entry: Before calling this routine you must push one 16 bit parameter onto the stack. This parameter is the FCB address.

On Return: FCBSTATE contains the result and the FCB address is popped from the stack.

DFTWRITECHAR (in PASIO) - This routine is used to write a character to a file. It is used by the WRITE and WRITELN Pascal constructs.

On Entry: Before calling this routine you must push three 16 bit parameters onto the stack in the following order:

1. FCB address
2. character (in low byte) to write
3. ASCII field size, number of ASCII bytes to write. The character is left-justified within this field.

On Return: FCBSTATE contains the result and the last two parameters are popped from the stack. The FCB address will remain on the stack.

DFTWRITEINT (in PASIO) - This routine is used to convert an integer to its ASCII equivalent and write it to a file. It is used by the WRITE and WRITELN Pascal constructs.

On Entry: Before calling this routine you must push three 16 bit parameters onto the stack in the following order:

1. FCB address
2. binary integer to write
3. ASCII field size, number of ASCII bytes to write. The number is right justified within this field.

On Return: FCBSTATE contains the result and the last two parameters are popped from the stack. The FCB address will remain on the stack.

DFTWRTREAL (in PASREAL) - This routine is used to convert a real number to its ASCII equivalent and write it to a file. It is used by the WRITE and WRITELN Pascal constructs.

On Entry: Before calling this routine you must push four 16 bit parameters onto the stack in the following order:

1. FCB address
2. address of the real number
3. ASCII field size, number of ASCII bytes to write. The number is right justified within this field. If the field is too small, then asterisks are output.
4. the number of decimal positions to the right of the decimal point. If this number is negative then scientific notation is used.

On Return: FCBSTATE contains the result and the last three parameters are popped from the stack. The FCB address will remain on the stack.

DFTWRITESTRG (in PASIO) - This routine is used to write a STRING to a file. It is used by the WRITE and WRITELN Pascal constructs.

On Entry: Before calling this routine you must push three 16 bit parameters onto the stack in the following order:

1. FCB address
2. address of the STRING
3. ASCII field size, number of ASCII bytes to write. The STRING is left-justified within this field.

On Return: FCBSTATE contains the result and the last two parameters are popped from the stack. The FCB address will remain on the stack.

DFTWRITETYPE (in PASIO) - This routine is used to write a number of bytes to a file. It is used by the WRITE Pascal construct.

On Entry: Before calling this routine you must push three 16 bit parameters onto the stack in the following order:

1. FCB address
2. address of the bytes to write
3. number of bytes to write

On Return: FCBSTATE contains the result and the last two parameters are popped from the stack. The FCB address will

remain on the stack.

DFTWRITELN (in PASIO) - This routine is used to write a carriage return to the file. It is used by the WRITELN Pascal construct.

On Entry: Before calling this routine you must push one 16 bit parameter onto the stack. This parameter is the FCB address.

On Return: FCBSTATE contains the result and the FCB address is popped from the stack.

DFTDISKREAD (in PASDISK) - This routine is used to read a sector from disk into the sector buffer of the FCB.

On Entry: The X register points to an FCB with the following fields filled in:

- FCBDEVNO contains the drive number
- FCBTRACK contains the track number
- FCBSECTR contains the sector number

On Return: All registers are preserved and the fields FCBSTATE and FCBBUFR have been filled in.

DFTDISKWRITE (in PASDISK) - This routine is used to write a sector to disk from the sector buffer of the FCB.

On Entry: The X register points to an FCB with the following fields filled in:

- FCBDEVNO contains the drive number
- FCBTRACK contains the track number
- FCBSECTR contains the sector number
- FCBBUFR filled in with the data to be written

On Return: All registers are preserved and the field FCBSTATE has been filled in.

DFTREADTAPE (in PASCASST) - This routine is used to read a block from cassette tape into the sector buffer of the FCB.

On Entry: The X register points to an FCB.

On Return: All registers are preserved and the following FCB fields are filled in:

- FCBSTATE contains the error condition
- FCBGRAN contains the block type
- FCBBUFSZ contains the size of the block read
- FCBBUFR contains the actual data
- FCBINDEX contains a zero

DFTWRITETAPE (in PASCASST) - This routine is used to write a block to cassette tape from the sector buffer of the FCB.

On Entry: The **B** register contains the block type and the **X** register points to an FCB with the following fields filled in:

- FCBBUFSZ contains the size of the block
- FCBBUFR filled in with the data to be written

On Return: All registers are preserved and the field FCBSTATE has been filled in. In addition, the field FCBBUFSZ is set to zero.

DFTPOLLKEY (in PASKEYBD) - This routine polls the keyboard to determine whether a key has been depressed. It performs the debounce and repeat functions.

On Entry: No entry conditions

On Return: All registers preserved except the **A** and CC registers. The **Z** bit in the CC register will be a 1 and the **A** register will equal zero if no character is present. If the **Z** bit is a 0, then the **A** register contains the ASCII character depressed.

DFTREADKEYBD (in PASKEYBD) - This routine runs the cursor, performs line editting and reads in an entire line's worth of data.

On Entry: The **X** register contains the FCB address.

On Return: All registers are preserved and the following fields are filled in:

- FCBINDEX is set to zero
- FCBBUFSZ is set to the size of the line read
- FCBBUFR contains the data
- FCBSTATE contains the resulting EOF status.

DFTSCREENOUT (in PASKEYBD) - This routine outputs a character to the screen.

On Entry: The A register contains the ASCII character to output.

On Return: All registers are preserved.

DFTRS232OUT (in PASKEYBD) - This routine outputs a character to the RS-232 port.

On Entry: The A register contains the ASCII character to output.

On Return: All registers are preserved.

6.2 General Utility Routines

DFTMULTIPLY (in PASRUNTM) - This routine is used to multiply two 16 bit values and return a 16 bit product upon return. The multiplication is performed using twos compliment binary arithmetic.

On Entry: Before calling, PSHS the two 16 bit numbers to be multiplied.

On Return: The D register contains the 16 bit result and all parameters are popped from the stack.

DFTDIVIDE (in PASRUNTM) - This routine is used to divide one 16 bit number into another 16 bit number. The division is performed using twos compliment binary arithmetic.

On Entry: Before calling, PSHS in the following order:

1. the 16 bit dividend
2. the 16 bit divisor

On Return: The dividend is replaced with the quotient and the divisor is replaced with the remainder

DFTHEX (in PASRUNTM) - This routine is used to convert binary data into a hexadecimal representation expressed in ASCII characters. The calling conventions are as follows:

On Entry:

- the X register contains the address in memory where the binary data which is to be converted resides

- the Y register contains the address of the area in memory which is to contain the the results of the HEX conversion
- the A register contains the number of bytes of binary data which are to be converted

On Return: The area in memory addressed by the contents of the Y register now contains the hexadecimal representation in ASCII form and all register contents are preserved.

DFTSTRUCTCMP (in PASRUNTM) - This routine compares two blocks of memory and sets the condition codes appropriately.

On Entry: Before calling, PSHS in the following order:

1. the address of the first memory block
2. the address of the second memory block
3. the number of bytes to compare

On Return: the first memory block is compared (logically subtracted) from the second memory block on a byte by byte basis. The condition codes are set appropriately, the parameters are popped from the stack.

DFTSTRCTLOAD (in PASRUNTM) - This routine loads an area of memory onto the stack.

On Entry: Before calling, PSHS in the following order:

1. the address of the memory area to load
2. the number of bytes to load

On Return: the parameters are popped from the stack and the memory area has been pushed onto the stack.

DFTSTRUCTMOV (in PASRUNTM) - This routine copies a block of memory.

On Entry: Before calling, PSHS in the following order:

1. the address of where you want the memory copied to
2. the address of where you want the memory copied from
3. the number of bytes to copy

On Return: the memory area is copied, the parameters are popped from the stack.

6.3 Real Number Routines

These routines provide the real arithmetic capability for Pascal. Remember that a real variable or constant is 6 bytes long but that an extra byte (for a total of 7) is always used when a real is loaded onto the stack in order to limit loss of precision.

DFTREALLOAD (in PASREAL) - This routine loads a real number onto the stack.

On Entry: the address of the real number to be loaded onto the stack is has been pushed on the stack.

On Return: the address has been popped and the real number has been loaded on the stack.

DFTREALSTORE (in PASREAL) - This routine stores a real number from the stack into a specific memory location.

On Entry: parameters are pushed onto the stack in the following order:

1. the address at which to store the real number.
2. the real number itself.

On Return: both the address and the real number are popped from the stack.

DFTREALNEG (in PASREAL) - This routine negates a real number on the stack.

On Entry: the real number to be negated is at the top of the stack.

On Return: the negated number is left on the stack.

DFTREALABS (in PASREAL) - This routine returns the absolute real value of a real number.

On Entry: the real number is at the top of the stack.

On Return: the absolute real number is left on the stack.

DFTREALFRACT (in PASREAL) - This routine returns the fractional portion of a real number.

On Entry: the real number is at the top of the stack.

On Return: the fractional real number is left on the stack.

DFTREALADD (in PASREAL) - This routine adds two real numbers.

On Entry: the two real numbers to add are at the top of the stack.

On Return: the two real numbers are replaced with their sum.

DFTREALSUB (in PASREAL) - This routine subtracts two real numbers.

On Entry: the subtrahend is at the top of the stack and the minuend is the next number down.

On Return: the two real numbers are replaced with their difference.

DFTREALMUL (in PASREAL) - This routine multiplies two real numbers.

On Entry: the two numbers to multiply are at the top of the stack.

On Return: the two real numbers are replaced with their product.

DFTREALDIV (in PASREAL) - This routine divides one real number by another.

On Entry: the divisor is at the top of the stack and the dividend is the next number down.

On Return: the two real numbers are replaced with their quotient.

DFTREALCMP (in PASREAL) - This routine compares one real number from another.

On Entry: the subtrahend is at the top of the stack and the minuend is the next number down.

On Return: the two real numbers are popped from the stack and the CC register is set accordingly.

DFTINTTOREAL (in PASREAL) - This routine converts an integer to a real.

On Entry: the 16 bit integer to be converted to a real is at the top of the stack.

On Return: the 16 bit integer is replaced with the corresponding real number.

6.4 String Routines

These routines operate on Pascal varying length strings. The first byte of the string is a length (0..255) with the remaining bytes containing the actual ASCII characters.

DFTSTRSTRCMP (in PASSTRNG) - This routine compares (logically subtracts) one string from another. The comparison is done a byte at a time until unequal bytes or the end of one of the strings is detected. If the end of a string is reached, then the string lengths are compared and the result returned.

On Entry: Before calling you must push two 16 bit parameters onto the stack in the following order:

1. the address of the subtrahend string.

2. the address of the minuend string.

On Return: both parameters are popped from the stack and the CC register is set accordingly.

DFTSTRSTRCPY (in PASSTRNG) - This routine copies one string to another. The length of the copy is determined by the length of the source string.

On Entry: Before calling you must push two 16 bit parameters onto the stack in the following order:

1. the address of the destination string.

2. the address of the source string.

On Return: both parameters are popped from the stack.

DFTSTRSTRAPP (in PASSTRNG) - This routine appends one string to another.

On Entry: Before calling you must push two 16 bit parameters onto the stack in the following order:

1. the address of the destination string.

2. the address of the source string.

On Return: both parameters are popped from the stack.

DFTCHRSTRCPY (in PASSTRNG) - This routine copies a character to a string.

On Entry: Before calling you must push two 16 bit parameters onto the stack in the following order:

1. the address of the destination string.
2. the character (in the low order byte) to be copied.

On Return: both parameters are popped from the stack.

DFTCHRSTRAPP (in PASSTRNG) - This routine appends a character to a string.

On Entry: Before calling you must push two 16 bit parameters onto the stack in the following order:

1. the address of the destination string.
2. the character (in the low order byte) to be appended.

On Return: both parameters are popped from the stack.

6.5 DEFT Object File Format

The object file is a sequential set of ASCII records produced by the compiler or assembler. These records all contain a single ASCII character type code, a set of parameters and a carriage return. The number and type of parameters required varies depending on type code. Following is a list of the codes and their parameter formats.

(A) Absolute Definition - Defines a global symbol with an absolute value. Two parameters are required.

symbol	**12**	**ASCII**
value	**4**	**HEX**

(B) Breakpoint - Used to specify that a breakpoint instruction ($3F) should be included if the Symbolic On-Line Debugger is included in the binary. If debug is not specified to the linker, then a NOP ($12) is generated. There are no parameters for this code.

(C) Comment - Used to specify a comment to be printed by the Linker on the link map. The single parameter contains the comment.

comment (1-254)		**ASCII**

(D) Define Storage - Used to reserve storage. A single parameter specifies the amount to reserve.

amount	**4**	**HEX**

(F) Fix-up - Used to fix-up a forward branch. The branch should come to the current location. The location of place to put the offset is the single parameter.

location	**4**	**HEX**

(J) Library Section - Marks the beginning of a library section in an object library. This will be the first record in an object library.

section name	**8**	**ASCII**

(K) Library Public Definition - Identifies a public symbol which is defined within this library section. After a **J** record, all the **K** records for a section will always immediately follow.

symbol name	**12**	**ASCII**

(L) Local Absolute Reference - Specifies the requirement for an absolute address generation. The single parameter specifies the offset that should be added to module's base address. Note that the presence of this code makes the resulting binary file non-PIC.

offset	**4**	**HEX**

(M) Main Entry Point - Specifies the place where execution should begin in the resulting binary file. This code has no parameters. Execution will begin at the point in the object where this type is encountered.

(P) Program Language Processor Marker - Identifies the language processor that produced the object file and provides debug and stack information for the linker and symbolic debugger.

language type	**1**	**ASCII**	**(P=Pascal, A=Assembler)**
stack req	**4**	**HEX**	
reserved	**1**	**ASCII**	
debug table off	**4**	**HEX**	

(R) Relative Definition - Used to define a global symbol which is relative to the beginning of the current module. The two parameters supply the symbol name and its offset from the current module.

symbol name	**12**	**ASCII**
offset	**4**	**HEX**

(S) Segment - Used to define a PIC segment of code. The parameter following the code is a variable length value containing the actual

code to insert.

code segment (2-254) HEX

(X) Absolute External Reference - Used to specify that a PUBLIC symbol is to have its absolute value placed here. The parameters specify the symbol name and offset to be added. Note that if the corresponding symbol was defined via an R then the resulting binary file is not PIC.

symbol name 12 ASCII
offset 4 HEX

(Y) Relative External Reference - Used to specify that a relative offset to a PUBLIC symbol is to placed here. The parameters specify the symbol name and offset to be added. Note that if the corresponding symbol was defined via an A then the resulting binary file is not PIC.

symbol name 12 ASCII
offset 4 HEX