The Compression Workshop

Version 1.00

CRESCENT SOFTWARE, INC.
11 BAILEY AVENUE
RIDGEFIELD, CT 06877
203-438-5300

LICENSE AGREEMENT

Crescent Software, Inc. grants a license to use the enclosed software and printed documentation to the original purchaser. Copies may be made for backup purposes only. Copies made for any other purpose are expressly prohibited, and adherence to this requirement is the sole responsibility of the purchaser. However, the purchaser does retain the right to sell or distribute programs that contain Compression Workshop subroutines in executable form only, so long as the primary purpose of the included routines is to augment the software being sold or distributed. Source code, object files, and libraries for any component of the Compression Workshop may not be distributed without prior arrangement with Crescent Software, Inc.

This license may be transferred to a third party only if all existing copies of the software and documentation are also transferred.

WARRANTY INFORMATION

Crescent Software, Inc. warrants that this product will perform as advertised. In the event that it does not meet the terms of this warranty, and only in that event, Crescent Software, Inc. will replace the product or refund the amount paid, if notified within 30 days of purchase and if the product was purchased directly from Crescent Software, Inc. Proof of purchase must be returned with the product, along with a brief description of how the product fails to meet the advertised claims.

CRESCENT SOFTWARE'S LIABILITY IS LIMITED TO THE PURCHASE PRICE. Under no circumstances shall Crescent Software, Inc. or the authors of this product be liable for any incidental or consequential damages, nor for any damages in excess of the original purchase price.

Table of Contents

1. Introduction

2. Reference

3. Compression Workshop Utilities

4. Technical Details

Chapter 1

Introduction

INTRODUCTION

INTRODUCTION

Thank you for purchasing the Compression Workshop. We have made every effort to create a powerful, yet easy to use set of tools for linking with Microsoft compiled BASIC. We sincerely hope that you find the Compression Workshop both useful and informative. If you have a comment, a complaint, or perhaps a suggestion for another BASIC-related product, please let us know. We want to be your *favorite* software company.

Before we begin discussing the contents of the Compression Workshop disk and manual, please take a few moments to fill out the enclosed registration card. Doing this entitles you to free technical support by phone, and ensures that you are notified of possible enhancements and new products. Many upgrades are offered at little or no cost, but we can't tell you about them unless we know who you are! Note, however, that if you purchased the Compression Workshop directly from us, the mail-in portion of the registration card may have been removed. In this case, you are already registered.

Also, please mark the Compression Workshop product serial number on your disk label or manual cover. License agreements and registration cards have an irritating way of becoming lost, and this helps you have the number handy if you need to contact us. You may also want to note the product version number in a convenient location. The version number is stored on the distribution disk in the volume label. If you ever have to call us for assistance, we will need to know your serial number, and probably the version you are using as well.

To determine the version number for any Crescent Software product, simply use the DOS VOL command, which displays the disk volume label:

```
VOL A:
Volume in drive A is CWShop 1.00
```

We are constantly improving all of our products, and you may want to call us periodically and ask for the current version number. Major upgrades are always announced, however minor additions or fixes are generally not. If you are having any problems at all—even if you are sure it is not caused by one of our products—please call us. We support all versions of compiled BASIC, and can often provide better assistance than Microsoft.

Overview

The Compression Workshop is comprised of four major components:

1. A set of subroutines and functions that can be added to your BASIC programs for compressing and decompressing array, string, file, and video memory data.

2. A full-featured Install utility you can use to distribute your own applications in a compressed format.

3. Subroutines to backup and restore disk data in a compressed format.

4. Demonstration programs that show how the Compression Workshop routines are used, and in many cases also serve as useful utilities.

The compression and decompression routines are designed to work with arrays, strings, and files. Numeric, TYPE, and fixed-length string arrays may be stored in compressed form on disk either individually, or in groups within a single file. One or more data and program files may also be combined into a single compressed file.

Arrays may be compressed either in place (within memory), or stored to a disk file. When storing a compressed array to a disk file you indicate if the file is to be created, or if the array data is to be added to an existing compressed file. Likewise, array data may be decompressed either in place or read from disk. When compressing an array in place, the compressed data replaces the original array contents, and the array is redimensioned to the new, smaller size automatically. Separate routines are used to manipulate data in memory and on disk.

String data may be compressed and decompressed in place only, and the result is assigned to the same string. Video memory may be copied to or from an array using the supplied routines, and then compressed and optionally stored on disk.

These routines cannot be used to compress or decompress entire conventional (not fixed-length) string arrays directly. However, if you have Crescent's QuickPak Professional library you can first use the StringSave routine to copy a string array to an integer array, and compress that. You would then use StringRestore afterward to copy the data back again after decompressing the integer array. QuickPak also lets you store compressed

data in either expanded or extended memory, to minimize the amount of memory your program needs to run.

A compressed file that holds other files (not one that holds arrays) may optionally be converted to a self-extracting .EXE program that will unpack itself when run, placing all of the files it contains into the current or any directory.

The backup and restore subprograms feature disk formatting, volume labeling, recursing nested subdirectory levels, and they can optionally manipulate each file's archive bit. They have been designed to automatically detect when the user changes disks, making them as easy to use as possible.

By default, Compression Workshop files have a .CWF extension, though you may of course use any other extension if you prefer. Note that compressed files which contain array data are different internally from compressed files that contain files. Therefore, you should consider using a different extension naming convention if there is any chance you might confuse a file's contents in your programs.

Finally, several demonstration programs are provided to show how the various Compression Workshop subroutines are used. These include utilities for compressing and decompressing groups of files based on command line options, an Install program that supports multiple distribution disks and multiple target paths, and a stand-alone hard disk backup program that also shows how the backup and restore routines can be added to your own programs.

All of the other BASIC demonstration programs have names that begin with the word DEMO, so you can easily identify them.

Installation

To install the Compression Workshop simply log onto your floppy drive and enter INSTALL at the DOS command prompt. On-screen instructions let you specify which drive and directory the product is to be installed to, and whether to install the assembly language source code. Note that you do not need to install the assembler source code to use the Compression Workshop. It is provided solely for the benefit of those people who are interested.

The default directory for installation is C:\CWSHOP, though you may of course change that if you prefer. If the destination directory does not yet exist INSTALL will create it for you.

Using The Compression Workshop

The Compression Workshop includes subroutines written using both BASIC and assembly language. The assembly language routines are furnished as Quick and LINK library files, and the BASIC routines are meant to be added to your programs by loading them as modules.

Using Integers

IMPORTANT: All numeric parameters must be integers except as noted. Integers are identified using a percent (%) suffix; for example, Variable% is an integer variable. Unless you are writing scientific or engineering programs that rely heavily on floating point (single and double precision) values, we recommend that you add the statement DEFINT A-Z as the very first line in your programs. Using DEFINT tells BASIC that all of your variables are to be integers unless you specify otherwise. You can then override this default as necessary when single precision, double precision, long integer, or Currency variables are needed. Currency variables are available in PDS 7.0 and later versions of Microsoft BASIC only.

Memory Allocation

Many of the Compression Workshop routines require a block of memory for use as a work space as they process your data. This memory is allocated by the various routines automatically, and it never exceeds 64K. Unless your program is using most of the PC's available memory for itself, this is not likely to cause a problem. Note that near memory (often called DGROUP) is never used by these routines, only far memory.

The exact amount of memory needed is listed at the beginning of each group of routines in the reference portion of this manual. However, this memory is not released automatically because the same memory is used repeatedly by all of the routines. Therefore, it is up to you to call CWReleaseMem when your program is no longer using these routines. As its name implies, CWReleaseMem releases the memory used by the various Compression Workshop routines, and makes it available to your BASIC program.

Please see the CWReleaseMem routine for more information on releasing memory.

Using Quick Libraries

To begin a QuickBASIC session and load the CWSHOP.QLB Quick Library, start QuickBASIC as follows:

```
QB [program] /L CWSHOP
```

If you are using BASIC 7 PDS or a newer version of Microsoft compiled BASIC, instead specify the CWSHOP7.QLB file like this:

```
QBX [program] /L CWSHOP7
```

As shown, you can optionally specify the name of a BASIC program that is also to be loaded.

IMPORTANT: A special version of the InterruptX routine is provided with the Compression Workshop. Unlike the standard version that comes with BASIC, this version was designed to accept only one instance of the Registers TYPE variable to improve efficiency. Please see the description of InterruptX elsewhere in this manual.

Loading Modules

Compression Workshop routines that are written in BASIC are intended to be loaded as modules into the Microsoft BASIC editor. You do this with the Load selection from the File menu. Once a source module has been loaded into the BASIC editor, you may call the subprograms and functions it contains.

When you subsequently save your main program, BASIC will create a Make file (having a .MAK extension) that contains the names of the modules your main program requires. Then when you use File/Open to open your main program later, the necessary support modules will also be loaded automatically.

Compiling And Linking

The Compression Workshop routines are provided in two LINK libraries, with one for use with QuickBASIC and BASIC 7 PDS when using near strings, and the other for use with BASIC PDS and later versions when compiling for use with far strings.

If you are using QuickBASIC or BASIC PDS with near strings, you should link your compiled BASIC programs with the CWSHOP.LIB library file as follows:

```
LINK PROGRAM [objfiles] , , NUL, CWSHOP.LIB ;
```

When creating a program that uses far strings you will instead link with the CWSHOP7.LIB file like this:

```
LINK PROGRAM [objfiles] , , NUL, CWSHOP7.LIB ;
```

There are a number of option switches that LINK recognizes, and the examples above merely show the minimum commands necessary to use each version of the Compression Workshop libraries. Refer to your BASIC manuals for information about other BC and LINK option switches.

- **QuickPak Professional**

The following routines from our QuickPak Professional product are included in CWSHOP.LIB and CWSHOP7.LIB for use by the backup and restore subprograms and also the CWPACK utility (InterruptX is from P.D.Q.):

DCount	ErrorMsg	Exist
FClose	FCount	FCreate
FGetA	FLof	FOpen
FormatDiskette	FPutA	GetAttr
GetDisketteType	GetVol	InterruptX
MakeDir	MidChar	PDQTimer
PutVol	ReadDir	ReadFile
SetAttr	WhichError	

Therefore, if you also have QuickPak you should link with the /noe switch to prevent LINK from reporting the duplicate names as an error:

```
LINK /NOE PROGRAM , , NUL, CWSHOP.LIB PRO.LIB ;
```

And if you are using BASIC 7 PDS or a later version with far strings do this:

```
LINK /NOE PROGRAM , , NUL, CWSHOP7.LIB PRO7.LIB ;
```

You can create a combined Quick Library that contains all of the routines from both products as follows:

```
LINK /Q CWSHOP.LIB PRO.LIB , BOTH.QLB , NUL, BQLB45.LIB ;
```

The BQLB45.LIB file comes with QuickBASIC 4.5. If you are using BASIC PDS or a later version you will instead specify CWSHOP7.LIB and PRO7.LIB as the main libraries, and use QBXQLB.LIB as the last argument on the LINK command line.

Note that creating a combined library this way will cause LINK to generate "Duplicate definition" errors because some routines are in both libraries. However, you can safely ignore these errors and the resulting Quick Library will still function as expected.

Error Codes

Most of the routines in the Compression Workshop return their success or failure in an integer error code parameter. If no error occurs, then the ErrCode% variable is returned holding a value of zero. Errors are indicated by one of the values listed in the table below.

ErrCode% = -1	Unable to allocate work area
ErrCode% = -2	Unable to create file
ErrCode% = -3	Unable to open file
ErrCode% = -4	File read/write error
ErrCode% = -5	Incorrect file ID
ErrCode% = -6	Unable to create temporary file
ErrCode% = -7	Unable to delete file
ErrCode% = -8	Input file too big to fit on one disk
ErrCode% = -9	Write attempt failed (disk full)
ErrCode% = -10	Critical error occurred
ErrCode% = -11	Incorrect time or date string length
ErrCode% = -12	Unable to open SELFEXT.EXE
ErrCode% = -13	No matching files found
ErrCode% = -14	Unable to compress
ErrCode% = -15	Array is not dynamic
ErrCode% = -16	Array ID not found
ErrCode% = -17	Array dimension mismatch
ErrCode% = -18	Bytes per element mismatch

NOTE: To reduce the number of arguments passed and thus reduce code size, the ErrCode% parameter is also used by some of the routines to pass a Mode value when they are called.

To simplify reporting errors you can use the ERRCODES.BI file. This is a BASIC Include file that dimensions an array named CWMsg$(), and assigns its elements to the appropriate text for each error message. This array is dimensioned with elements numbered -18 through 0, so you can directly use the values returned in the ErrCode% variable:

```
IF ErrCode% THEN PRINT CWMsg$(ErrCode)
```

Note error number -10, which indicates that a DOS critical error occurred. Critical errors are those that result in the infamous "Abort/Retry/Fail" DOS message, and they occur when a disk is not in the drive, or when the disk has not been formatted. Unfortunately, DOS handles critical errors differently from other errors, and uses a different numbering system for them.

If a critical error occurs you can determine which one with the CWCritErr% function. All of the DOS critical errors that can be returned by the Compression Workshop routines are also defined in the ERRCODES.BI file in the CWCritMsg$() string array. The standard DOS critical error codes are as follows:

CWCritErr% = 0	Disk write-protected
CWCritErr% = 1	Invalid drive
CWCritErr% = 2	Drive not ready
CWCritErr% = 3	Unknown command
CWCritErr% = 4	CRC data error
CWCritErr% = 5	Bad request length
CWCritErr% = 6	Seek error
CWCritErr% = 7	Unknown disk format
CWCritErr% = 8	Sector not found
CWCritErr% = 9	Printer out of paper (not possible here)
CWCritErr% = 10	Write fault
CWCritErr% = 11	Read fault
CWCritErr% = 12	Other error
CWCritErr% = 15	Invalid disk change (DOS 3.0 or later only)

The following code shows the steps needed to properly report every type of error that can occur when using the Compression Workshop routines:

```
IF ErrCode% THEN
    PRINT "Error"; ErrCode%; "occurred: ";
    PRINT CWMsg$(ErrCode%)
    IF ErrCode% = -10 THEN
      PRINT " -- "; CWCritMsg$(CWCritErr%)
    END IF
END IF
```

Subroutine Declarations

BASIC does not require you to declare called subroutines, though doing this can help you to avoid system crashes caused by using the wrong number or type of parameters. However, functions must be declared whether they are written in BASIC or assembly language.

The CWDECL.BI file contains declarations for all of the Compression Workshop subroutines and functions, and it is meant to be added to your programs using the '$INCLUDE metacommand. If you are using both ERRCODES.BI and CWDECL.BI in the same program you must list CWDECL.BI first, because BASIC does not allow executable code to be placed before DECLARE statements. (ERRCODES.BI contains code that dimensions and assigns the CWMsg$() and CWCritMsg$() string arrays.)

Array Redimensioning

Many Compression Workshop routines accept as parameters entire arrays that are passed using empty parentheses. Some of these routines, such as CWReadNames and CWUnpackArray, also redimension the arrays to the correct number of elements to hold the returned information. Therefore, before calling these routines you must establish the array using REDIM—generally with only one element—to ensure that the array is dynamic.

The following is from the syntax example for CWReadNames elsewhere in this manual:

```
REDIM CW(1 TO 1) AS CWType
CALL CWReadNames(FileName$, CW(), ErrCode%)
PRINT "There are"; UBOUND(CW); "files present."
```

Note that all of the routines that redimension arrays honor the starting element number in effect at the time they are called. Here, the array was established with element one as the first element, so the value UBOUND returns corresponds to the number of file names present. Had the array been established using element zero as the first element, then the actual number of file names would be one less:

```
REDIM CW(0) AS CWType
CALL CWReadNames(FileName$, CW(), ErrCode%)
PRINT "There are"; UBOUND(CW) - 1; "files present."
```

INTRODUCTION

Chapter 2

Reference

REFERENCE

DYNAMIC ARRAY ROUTINES

The Compression Workshop includes a number of routines that operate on BASIC dynamic arrays, both in memory and in disk files. Routines having names that end with the letter M compress and decompress array data in memory; the remaining routines operate on arrays that are stored in disk files.

The dynamic array routines require 46,016 bytes of far memory to be available to them for use as a work space. This memory is allocated automatically by the various Compression Workshop array routines, but it is up to you to call CWReleaseMem (using a Mode value of 1) to release the memory when you no longer need it. See the description for CWReleaseMem for more information.

When compressing array data to a disk file, you assign an ID number that will be used to subsequently identify each array in the file. This number can be any integer value between -32768 and 32767; therefore, as many as 65,536 arrays may be stored in a single compressed file.

Arrays are stored in a disk file one at a time, and you specify whether the file is to be created or appended to. Therefore, to store several arrays you will first call CWPackArray asking it to create a new file, and then call it repeatedly for each additional array telling it to append to the same file. This is shown in the CWPackArray description that follows.

Note that updating an existing array in a compressed file which contains more than one array is a two-step process: First the existing version is deleted with the CWDelArray routine, and then the new version of the array is added using CWPackArray to append it to the same file.

Using Fixed-length String Arrays

To process fixed-length string arrays the array must be created as a TYPE array as follows:

```
TYPE FLen
  A AS STRING * LL                    'LL = desired length
END TYPE
REDIM Array(NN) AS FLen               'NN = desired number of elements
```

You cannot pass a fixed-length string array directly using empty parentheses.

CWDelArray (subroutine)

- Purpose:

  Deletes an array from a compressed file.

- Syntax:

```
CALL CWDelArray(FileName$, ArrayID%, ErrCode%)
```

- Where:

FileName$:	Name of an existing compressed file
ArrayID%:	Unique array identifier
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

  See the DEMOARAY.BAS demonstration program for an example of using CWDelArray.

CWGetDetails (subroutine)

- Purpose:

  CWGetDetails fills an integer array with the number of elements contained in the specified array. An array is used to return the information because the compressed arrays may have more than one dimension. CWGetDetails is meant to be used in conjunction with CWReadIDs, to obtain complete information about the arrays stored in a compressed file.

- Syntax:

```
CALL CWGetDetails(FileName$, ArrayID%, Details%(), ErrCode%)
```

- Where:

FileName$:	Name of an existing compressed file
ArrayID%:	Valid array ID (which array to report on)
Details%():	Integer array in which to return the compressed array details
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

The array passed as Details%() must be properly sized, since it is not redimensioned by CWGetDetails. The correct number of elements is determined by first calling CWReadIDs:

```
REDIM Details%(1 to AInfo(X).ArrayDimensions)
```

See the description for CWReadIDs elsewhere in this manual, and also see the DEMOARA2.BAS example program.

CWReadIDs (subroutine)

- Purpose:

  CWReadIDs fills a TYPE array with information about all of the arrays present in a compressed file.

- Syntax::

```
CALL CWReadIDs(FileName$, AInfo(), ErrCode%)
```

- Where:

FileName$:	Name of an existing compressed file
AInfo():	Special TYPE array that returns the array information
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

The array passed as AInfo() is constructed as follows:

```
TYPE ArrayInfo
  ArrayID AS INTECER
  BytesPerElement AS INTEGER
  ArrayDimensions AS INTEGER
END TYPE
REDIM AInfo(1 TO 1) AS ArrayInfo
```

The array can be any size, as it will be redimensioned to the proper number of elements by CWReadIDs.

After running DEMOARAY.BAS to create a compressed file containing array information, run DEMOARA2.BAS to see how CWReadIDs can be used.

CWPackArray (subroutine)

- Purpose:

  Compresses an array and saves it in a compressed file.

- Syntax:

```
CALL CWPackArray(Array(), ArrayID%, FileName$, ErrCode%)
```

- Where:

Array():	Array to compress, empty parentheses are required
ArrayID%:	Unique array identifier you assign
FileName$:	Name of compressed file to create

ErrCode% (when calling):

Zero = Create new compressed file (overwrite any existing)
Non-zero = Append to existing compressed file

ErrCode% (upon return):

Return value as listed in the section *Error Codes*

- Comments:

The value of ErrCode% when the call is made determines whether CWPackArray will create a new compressed file or append to an existing compressed file. Upon return, ErrCode% indicates the success or failure of the operation, as explained in the section *Error Codes*.

See the DEMOARAY.BAS example program.

CWPackArrayM (subroutine)

- Purpose:

  Compresses an array in place (in memory).

- Syntax:

```
CALL CWPackArrayM(Array(), ErrCode%)
```

- Where:

Array():	Array to compress, the empty parentheses are required
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

  The array will be redimensioned to a smaller size, and then filled with the compressed data.

CWUnpackArray (subroutine)

- Purpose:

  Reads and unpacks an array contained in a compressed file.

- Syntax:

```
CALL CWUnpackArray(Array(), ArrayID%, FileName$, ErrCode%)
```

- Where:

Array():	Array to decompress, the empty parentheses are required
ArrayID%:	Unique array identifier (which array to decompress)
FileName$:	Name of an existing compressed file
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

The type of the array and whether it is multi-dimensional must be known by the programmer. The array will be redimensioned to the proper number of elements, but you must at least establish the number of dimensions and also ensure the array is dynamic using either REDIM or '$DYNAMIC.

For example, if you used CWPackArray to save an array that was dimensioned using this:

```
REDIM Array%(1 TO 10, 1 TO 100)
```

then you should use REDIM before calling CWUnpackArray like this:

```
REDIM Array%(1 TO 1, 1 TO 1)
```

Note that any starting array element number may be used, and that element number will be retained by CWUnpackArray. So if the array was dimensioned originally using **REDIM (1 TO 10)** and then saved using CWPackArray, using **REDIM (0)** before calling CWUnpackArray will create the array with elements (0 TO 9) instead of (1 TO 10).

See the DEMOARAY.BAS example program.

CWUnpackArrayM (subroutine)

- Purpose:

  Decompresses an array in place (in memory).

- Syntax:

```
CALL CWUnpackArrayM(Array(), ErrCode%)
```

- Where:

  Array(): Previously compressed array
  ErrCode%: Return value as listed in the section *Error Codes*

- Comments:

  The array will be redimensioned to the original number of elements and then decompressed.

STRING ROUTINES

The string routines require 46,016 bytes of far memory to be available to them for use as a work space. This memory is allocated automatically by the Compression Workshop string routines, but it is up to you to call CWReleaseMem (using a Mode value of 1) to release the memory when you no longer need it. See the description for CWReleaseMem for more information.

Strings must be at least eight characters long to be compressed. Further, in order to be compressed a string must contain repeated characters. Therefore, longer strings are more likely to be compressed effectively.

Note that you cannot retrieve a compressed string that was written to disk using INPUT or LINE INPUT. These commands expect to find a CHR$(13) carriage return in the file, to indicate the end of the string. Since compressing a string may happen to imbed a CHR$(13) within it, BASIC will stop reading at that point, and also discard the byte. If you plan to store compressed strings on disk, we recommend that you precede each one with a length word in the file. This way you can retrieve the string using Binary file access and the GET command.

The following example compresses and writes two strings to a disk file, and then reads them back again and decompresses them:

```
OPEN "TEST.DAT" FOR BINARY AS #1          'open for Binary

CALL CWPackStringM(S1$, ErrCode%)         'pack this string
Length% = LEN(S1$)                        'get its length
PUT #1, , Length%                         'first write the length
PUT #1, , S1$                             'then write the string

CALL CWPackStringM(S2$, ErrCode%)         'as above
Length% = LEN(S2$)
PUT #1, , Length%
PUT #1, , S2$

CLOSE #1                                  'close the file


OPEN "TEST.DAT" FOR BINARY AS #1          'open the file again

GET #1, , Length%                         'first get the length
S1$ = SPACE$(Length%)                     'create the string
```

```
GET #1, , S1$                          'then read it from disk
CALL CWUnPackStringM(S1$, ErrCode%)    'finally unpack it

GET #1, , Length%                      'as above
S2$ = SPACE$(Length%)
GET #1, , S2$
CALL CWUnPackStringM(S2$, ErrCode%)
```


Two strings were used to show that you can easily walk through the file to retrieve them in succession, much as you would with INPUT or LINE INPUT. Binary file operations use the length of the destination or source variable to know how many bytes are to be read or written respectively. If you plan to write and read many strings, you can use the following BASIC subprograms to do this more efficiently:

```
SUB PutString(Work$, FileNumber%, ErrCode%)
 CALL CWPackStringM(Work$, ErrCode%)
 IF ErrCode% THEN EXIT SUB
 Length% = LEN(Work$)
 PUT #FileNumber%, , Length%
 PUT #FileNumber$, , Work$
END SUB

SUB GetString(Work$, FileNumber%, ErrCode%)
  GET #1, , Length%
  Work$ = SPACE$(Length%)
  GET #FileNumber%, , Work$
  CALL CWUnPackStringM(Work$, ErrCode%)
END SUB
```

You can either use BASIC's EOF function to determine when you have read all of the strings in a Binary file, or store the number of strings at the very beginning of the file. In that case you would first use GET to read the total number of strings, and then use a FOR/NEXT loop to read each string. This method is ideally suited for reading entire string arrays:

```
OPEN "TEST.DAT" FOR BINARY AS #1
GET #1, , NumStrings%
REDIM Array$(1 TO NumStrings%)

FOR X = 1 TO NumStrings%
  CALL GetString(Array$(X), 1, ErrCode%)
  IF ErrCode% THEN EXIT FOR
NEXT

CLOSE #1
IF ErrCode% THEN PRINT "Error reading string #"; X
```

CWPackStringM (subroutine)

- Purpose:

  Compresses a string in place (in memory).

- Syntax:

```
CALL CWPackStringM(Work$, ErrCode%)
```

- Where:

  Work$: String to be compressed
  ErrCode%: Return value as listed in the section *Error Codes*

- Comments:

  The string is resized to a shorter length, and then filled with a compressed version of its original contents.

  Strings less than eight characters long cannot be compressed, and attempting to do that will result in an "Unable to compress" error (-14).

CWUnpackStringM (subroutine)

- Purpose:

  Decompresses a string in place (in memory).

- Syntax:

```
CALL CWUnpackStringM(Work$, ErrCode%)
```

- Where:

  Work$: Previously compressed string
  ErrCode%: Return value as listed in the section *Error Codes*

- Comments:

  The string will be restored to its original length and contents.

  Be careful not to call CWUnpackStringM with a string that is not already compressed. Depending on the string's contents, BASIC might crash with an "Out of string space" error if falsely decompressing the data creates a very long string in the process.

FILE ROUTINES

These routines operate on files that already exist, and let you compress one or more of them storing the result into a single compressed file. When compressing files the original files are never altered. Rather, they are compressed in memory, and then either added to a newly created file or appended to an existing compressed file.

Compressed files may contain an optional comment string, to help identify their contents. In the supplied INSTALL.BAS program, the compressed file comments are used to indicate the default destination directories.

The file routines require 56,016 bytes of far memory to be available to them for use as a work space. This memory is allocated automatically by the Compression Workshop routines, but it is up to you to call CWRelease-Mem (using a Mode value of 0) to release the memory when you no longer need it. See the description for CWReleaseMem for more information.

The names of all files that are being added to a compressed file may include a full path and drive, up to a maximum length of 67 characters.

An important feature of the Compression Workshop file routines is the ability to combine multiple disk files into more than one compressed file, to allow them to be fit onto floppy disks. You can create multiple compressed files from a single file specification in one of two ways:

1. If the destination disk becomes full while creating a compressed file, the file compression routines will return an ErrCode% value of -9. You will then leave ErrCode% set to -9 and call the file compression routine again, after prompting the user to insert a new disk and enter a new file name.

2. If you are creating files on a hard disk with the intention of copying them to a floppy disk later, you can use the CWSetMax-Size routine to force a "Disk full" error when the file reaches a given size. You will then call the routine again with ErrCode% left set to -9, specifying a different name for the next consecutive compressed file to be created.

Note that compression and decompression speeds are greatly affected by disk access times. Thus, using a hard disk or RAM drive is considerably better than using a floppy disk.

This page intentionally left blank.

CWDelFile (subroutine)

- Purpose:

  Deletes a file from within an existing compressed file.

- Syntax:

```
CALL CWDelFile(FileName$, Spec$, ErrCode%)
```

- Where:

FileName$:	Name of an existing compressed file
Spec$:	File(s) to be deleted, may contain the (*) wildcard, the (?) wildcard is not supported
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

  See the CWPACK.BAS program for an example of calling CWDelFile.

CWGetComment (subroutine)

- Purpose:

  Returns a compressed file's comment string.

- Syntax:

```
CALL CWGetComment(FileName$, Cmt$, ErrCode%)
```

- Where:

FileName$:	Name of an existing compressed file
Cmt$:	Returns holding the compressed file's comment string
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

  If ErrCode% is returned set to zero and LEN(Cmt$) = 0, then the file does not contain a comment string.

CWMakeExt (subroutine)

- Purpose:

  Creates a self-extracting .EXE program from a compressed file.

- Syntax:

```
CALL CWMakeExt(FileName$, Buffer$, ErrCode%)
```

- Where:

FileName$:	Name of an existing compressed file
Buffer$:	String to place into keyboard buffer (9 characters maximum)
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

A new file with the same name as the compressed file but an .EXE extension is created in the directory where the compressed file resides. The new file is a composite of SELFEXT.EXE (supplied with the Compression Workshop) and the compressed file. When the self-extracting program is run it "unpacks" itself leaving the files it contains in the current directory. Comments contained in the original compressed file are echoed to the screen when the self-extracting program file executes.

The compressed file parameter may include a path, but SELFEXT.EXE must be in the same directory as the executing program. This is BASIC's directory when in the BASIC editing environment. If you are using QuickBASIC and running QB.EXE from a directory other than the one in which it resides, you will need to place a copy of SELFEXT.EXE in the directory that holds QB.EXE.

The contents of Buffer$ is placed into the PC's type-ahead buffer when the self-extracting program file finishes running. This lets you run another program or batch file automatically.

To start another program use:

```
Buffer$= "PROGNAME" + CHR$(13)
```

If Buffer$ is null, no command is placed into the keyboard buffer.

IMPORTANT: CWMakeExt requires DOS 3.0 or later both when creating a self-extracting file and also when that file is run.

CWPackedSize (function)

- Purpose:

  CWPackedSize returns the new size a file will become when it is added into a compressed file.

- Syntax:

```
NewSize& = CWPackedSize&(FileName$, ErrCode%)
```

- Where:

FileName$:	Name of the file to test
ErrCode%:	Return value as listed in the section *Error Codes*
NewSize&:	New size of the file after compression

- Comments:

  Because CWPackedSize has been designed as a function, it must be declared before it may be used.

  CWPackedSize lets you know how small a file will become when it is compressed later, and it is meant to help you avoid potential "Disk full" errors before they occur.

  The value returned by CWPackedSize includes the size of the internal file header, but not the main compressed file header (six bytes + comment length). Therefore, if you are going to compress only one file you should add 6 plus the comment length, to determine the exact size the compressed file will be.

  CWPackedSize works by actually compressing the file on a trial basis, and then seeing how large the result is. Therefore, you should call CWReleaseMem after using this routine if your program needs all available memory.

CWPackFiles (subroutine)

- Purpose:

  CWPackFiles compresses all of the files that match the search specification and stores them in the destination file.

- Syntax:

```
CALL CWPackFiles(Spec$, Dest$, Cmt$, ErrCode%)
```

- Where:

Spec$:	Specification for file(s) to compress, may include a path, accepts both DOS wildcards (* and ?)
Dest$:	Name of the destination compressed file
Cmt$:	Comment string for the compressed file (optional)

ErrCode% (when calling) :

Zero = Create a new compressed file (overwrite any existing)
Non-zero = Append to an existing compressed file
-9 = Continue the last operation

ErrCode% (upon return) :

Return value as listed in the section *Error Codes*

- Comments:

The value of ErrCode% when the call is made determines whether this routine will create a new compressed file or append to an existing compressed file. When appending to an existing file the comment string (Cmt$) is ignored if present. A comment string can be as long as BASIC allows—up to 32,767 characters.

Upon return, ErrCode% indicates the success or failure of the operation, as explained in the section *Error Codes*. The section that follows lists some of the possible reasons for errors.

ErrCode% = -9 (file write failed) will occur if the disk is full, or if you specified a maximum file size using CWSetMaxSize and that limit was reached. In either case, your program should prompt the user to insert another disk. Then call CWPackFiles again with a new compressed file name, leaving ErrCode% set to -9. The same file specification will be

used for the new compressed file, and processing will resume with the same input file that did not fit in the previous compressed file. If a single file is larger than the amount of disk space available it cannot be compressed with this routine.

Note that the first compressed file that is created must not be in the same directory as the search specification if "*.*" was used, because it will be added into the second compressed file.

It is not possible to compress a file and add it to a compressed file having the same name. Further, files must be at least two bytes long to be added to a compressed file.

When using the append mode with a "*.*" file specification, the calling program must ensure that files already in the existing compressed file won't be found, because this would put a second copy of the files into the destination compressed file.

CWPackFilesD (subroutine)

- Purpose:

  Compresses all files that match the search specification and are dated later (newer) than the time/date specified, and adds them to the destination file.

- Syntax:

```
CALL CWPackFilesD(Spec$, Dest$, Cmt$, Tim$, Dat$, ErrCode%)
```

- Where:

Spec$:	Specification for file(s) to compress, may include a path, accepts both DOS wildcards (* and ?)
Dest$:	Name of the destination compressed file
Cmt$:	Comment string for compressed file
Tim$:	8-character time string HH:MM:SS (military time)
Dat$:	10-character date string MM-DD-YYYY

ErrCode% (when calling) :

Zero = Create a new compressed file (overwrite any existing)
Non-zero = Append to an existing compressed file
-9 = Continue the last operation

ErrCode% (upon return) :

Return value as listed in the section *Error Codes*

- Comments:

The value of ErrCode% when the call is made determines whether this routine will create a new compressed file or append to an existing compressed file. When appending to a file the comment parameter (Cmt$) is ignored if present. A comment string can be as long as BASIC allows—up to 32,767 characters.

Upon return, ErrCode% indicates the success or failure of the operation, as explained in the section *Error Codes*. The section that follows lists some of the possible reasons for errors.

ErrCode% = -9 (file write failed) will occur if the disk is full, or if you specified a maximum file size using CWSetMaxSize and that limit was reached. In either case your program should prompt the user to insert

another disk. Then call CWPackFilesD again with a new compressed file name, leaving ErrCode% set to -9. The same file specification will be used for the new compressed file, and processing will resume with the same input file that did not fit in the previous compressed file. If a single file is larger than the amount of disk space available it cannot be compressed with this routine.

Note that the first compressed file that is created must not be in the same directory as the search specification if "*.*" was used, because it will be added into the second compressed file.

It is not possible to compress a file and add it to a compressed file having the same name.

When using the append mode with a "*.*" file specification, the calling program must ensure that files already in the existing compressed file won't be found, because this would put a second copy of the files into the destination compressed file.

The date and time arguments must be given exactly as shown. See the FixDate and FixTime functions elsewhere in this manual; these can be used to ensure that the date and time formats are consistent with what is expected.

CWReadNames (subroutine)

- Purpose:

  Fills a TYPE array with the names, dates, times, and original sizes of all the files stored in a compressed file.

- Syntax:

```
CALL CWReadNames(FileName$, TypeArray(), ErrCode%)
```

- Where:

FileName$:	Name of an existing compressed file
TypeArray():	A special TYPE array that returns the name, date, time, and original size of the individual files (see below)
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

The array passed as TypeArray() is constructed as follows:

```
TYPE CWType
  FileName AS STRING * 12
  FileDate AS STRING * 3    'CHR$(Year) + CHR$(Month) + CHR$(Day)
  FileTime AS STRING * 3    'CHR$(Hour) + CHR$(Minute) + CHR$(Second)
  FileSize AS LONG
END TYPE
```

The array can be any size, as it will be redimensioned to the proper size by CWReadNames. However, the array must be dynamic so its size can be changed.

You can therefore determine how many files are present in a compressed file by calling CWReadNames to read their names, and then examine the upper bound using UBOUND like this:

```
REDIM CW(1 TO 1) AS CWType
CALL CWReadNames(FileName$, CW(), ErrCode%)
PRINT "There are"; UBOUND(CW); "files present."
```

The time and date are returned as shown above to facilitate comparing them. For example, you could sort a list of files based on their dates using the following code, perhaps within a bubble sort:

```
IF CW(X).FileDate > CW(X + 1).FileDate THEN
  SWAP CW(X), CW(X + 1)
END IF
```

And to compare based on both the file dates and times you could concatenate the strings like this:

```
IF CW(X).FileDate + CW(X).FileTime >_
  CW(X + 1).FileDate + CW(X + 1).FileTime THEN
    SWAP CW(X), CW(X + 1)
END IF
```

CWReleaseMem (subroutine)

- Purpose:

  Releases memory allocated by the compression and decompression routines.

- Syntax:

```
CALL CWReleaseMem(Mode%)
```

- Where:

  Mode% = 0: Release memory allocated by the file routines
  Mode% = 1: Release memory allocated by the array and string routines

- Comments:

Only routines that compress and decompress allocate memory. The first call made to one of those routines will allocate the required memory. BASIC's far heap is reduced making the memory accessible only to subsequent calls to the compression and decompression routines. When the memory is needed for other use it should be released by calling CWReleaseMem.

It is not strictly necessary to call CWReleaseMem prior to ending a program once it has been compiled, since DOS will reclaim the memory when the program ends anyway. However, if the program is run in the BASIC editor and it ends without releasing the memory, the next time the program is run that much less memory will be available. After running the program a few times you'll then receive an "Out of memory" error. If that happens you should quit BASIC and restart it again.

Although you could press F6 to get to the immediate window and enter **CALL CWReleaseMem(Mode)**, that relinquishes only the most recently allocated memory. Any memory that was used by the Compression Workshop from earlier runs will not be returned to BASIC.

IMPORTANT: If you are using both the memory and file compression routines you must call CWReleaseMem twice—once with a Mode value of zero and once again with a Mode value of one.

CWSetMaxSize (subroutine)

- Purpose:

  Sets the maximum compressed file size for the CWPackFiles and CWPackFilesD routines.

- Syntax:

```
CALL CWSetMaxSize(MaxSize&)
```

- Where:

  MaxSize&: Long integer value

- Comments:

  CWSetMaxSize is useful for ensuring that a compressed file being created on a hard disk will fit when copied to a floppy disk. Of course, you can simply compress a series of files and then use DIR to see how large the resulting compressed file is. However, CWSetMaxSize lets you conveniently create files that are guaranteed to fit. If the size you specify is reached and there are still more input files to be added, you can call CWPackFiles or CWPackFilesD again to continue adding them to other, subsequent compressed files.

  The default is no limit on compressed file size. You can use this routine to force ErrCode to -9 when a compressed file reaches the limit. Once set, this limit is honored by all of the compression routines. To remove a previously set limit call CWSetMaxSize with MaxSize& set to 0.

CWUnpackFiles (subroutine)

- Purpose:

  Decompresses all files that match a given search specification.

- Syntax:

```
CALL CWUnpackFiles(FileName$, Spec$, ErrCode%)
```

- Where:

FileName$:	Name of the compressed file
Spec$:	Specification for the file(s) to decompress, may include a path, may contain the (*) wildcard, the (?) wildcard is not supported
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

  All files found in the compressed file that match the given specification are decompressed. Any existing files will be overwritten.

  If a path name precedes the file specification, the files are placed there instead of in the current directory. This path name may of course contain a drive letter, a directory name, or both.

CWUnpackFilesD (subroutine)

- Purpose:

  Decompresses files that match the search specification overwriting only older versions.

- Syntax:

```
CALL CWUnpackFilesD(FileName$, Spec$, ErrCode%)
```

- Where:

FileName$:	Name of the compressed file
Spec$:	Specification for the file(s) to decompress, may include a path, may contain the (*) wildcard, the (?) wildcard is not supported
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

  All files found in the compressed file that match the specification and are either not present or are newer than any existing files in the current or stated directory will be decompressed.

  If a path name precedes the file specification, the files are placed there instead of in the current directory.

  You can optionally specify that only files that are newer and also already exist be unpacked by setting ErrCode% to any non-zero value before calling CWUnpackFilesD. This is useful for distributing product updates, where you originally allowed the user to install only selected files. When updating the files with the newer versions, only those files that has already been installed will be updated.

CWUpdate (subroutine)

- Purpose

  Updates one or more files in an existing compressed file.

- Syntax:

```
CALL CWUpdate(FileName$, TmpFile$, ErrCode%)
```

- Where:

FileName$:	Name of an existing compressed file
TmpFile$:	Name of the temporary work file
ErrCode%:	Return value as listed in the section *Error Codes*

- Comments:

The current directory is searched for newer files that correspond to files already present in the compressed file. The compressed file is then updated with those files. If the new files are located in another directory then you should change to that directory using BASIC's CHDIR command, and specify the full path to the compressed file being processed.

The TmpFile$ file name specifies a temporary work file that CWUpdate needs as it processes the main compressed file. This temporary file is used to build a new compressed file holding the updated versions of each internal file. It is then copied over the original compressed file when CWUpdate is finished, and the temporary file is deleted.

The temporary file name may contain a drive and path if you want, perhaps to specify a RAM drive for faster processing. When processing a compressed file that is on a floppy disk, you may want to use a hard disk to prevent a "Disk full" error if the floppy does not have enough room for both the original and temporary files.

Some users set a TMP or TEMP environment variable to specify where temporary files such as this are to be written. The CWPACK.BAS utility shows how to look for such environment variables, and use that drive and path if present.

DISPLAY MEMORY ROUTINES

This page intentionally left blank.

CWArray2Scrn (subroutine)

- Purpose:

  Copies a BASIC dynamic array to the text screen or other memory area.

- Syntax:

```
CALL CWArray2Scrn(Segment%, Address%, NumBytes%, Array())
```

- Where:

Segment%:	&HB800 for a color text screen, &HB000 for monochrome, or any arbitrary segment
Address%:	The address within Segment%, usually zero
NumBytes%:	Number of bytes to copy, use a long integer for values between 32,767 and 65,535
Array():	Source array, must be dynamic

- Comments:

  This routine complements CWScrn2Array, and it is used to restore the data to the original location in screen memory.

CWScrn2Array (subroutine)

- Purpose:

  Copies a text screen or other area of memory to a BASIC dynamic array.

- Syntax:

```
CALL CWScrn2Array(Segment%, Address%, NumBytes%, Array())
```

- Where:

Segment%:	&HB800 for a color text screen, &HB000 for monochrome, or any arbitrary segment
Address%:	The address within Segment%, usually zero
NumBytes%:	Number of bytes to copy, use a long integer for values between 32,767 and 65,535
Array():	Destination array, must be dynamic

- Comments:

The routines in the Compression Workshop expect arrays as parameters, rather than memory segments and addresses. This routine lets you copy text or graphics screens (or nearly any memory area) to a BASIC dynamic array, which can in turn be compressed. Note that you must properly dimension the array to receive the screen or memory contents before calling CWScrn2Array.

CRITICAL ERROR HANDLING

This page intentionally left blank.

CWCritErr (function)

- Purpose:

  Returns the DOS critical error code after a critical error occurred.

- Syntax:

```
CritCode = CWCritErr%
```

- Where:

  CritCode receives one of the standard DOS critical error codes shown below:

CritCode = 0	Disk write-protected
CritCode = 1	Invalid drive
CritCode = 2	Drive not ready
CritCode = 3	Unknown command
CritCode = 4	CRC data error
CritCode = 5	Bad request length
CritCode = 6	Seek error
CritCode = 7	Unknown disk format
CritCode = 8	Sector not found
CritCode = 9	Printer out of paper (not possible here)
CritCode = 10	Write fault
CritCode = 11	Read fault
CritCode = 12	Other error
CritCode = 15	Invalid disk change (DOS 3.0 or later only)

- Comments:

  Because CWCritErr has been designed as a function, it must be declared before it may be used.

  When any of the routines in this library return a value of -10 for ErrCode, you should call CWCritErr to retrieve the actual DOS critical error code.

  See the section *Error Codes* for more information about DOS critical errors.

This page intentionally left blank.

TIME AND DATE FORMATTING ROUTINES

This page intentionally left blank.

FixDate (BASIC function)

- Purpose:

  FixDate accepts a date in a variety of formats, and returns the same date in a form that is consistent with what CWPackFilesD expects.

- Syntax:

```
FixedDate$ = FixDate$(AnyDate$)
```

- Where:

  AnyDate$ can be in the form MM-DD-YY or M/D/YYYY or any such combination. FixedDate$ then receives the date in the form MM-DD-19YY.

- Comments:

  Because FixDate has been designed as a function, it must be declared before it may be used. Further, FixDate is a BASIC function contained in the DATETIME.BAS file. This file must be loaded as a module and later linked with your main BASIC program.

  FixDate$ lets you accept a date from the user and use that as an argument to CWPackFilesD, confident that the information is in the correct format.

FixTime (BASIC function)

- Purpose:

  FixTime accepts a time in a variety of formats, and returns the same time in a form that is consistent with what CWPackFilesD expects.

- Syntax:

```
FixedTime$ = FixTime$(AnyTime$)
```

- Where:

  AnyTime$ can be in the form HH:MM:SS or H-M-S or any such combination. FixedTime$ then receives the time in the form HH:MM:SS.

- Comments:

  Because FixTime has been designed as a function, it must be declared before it may be used. Further, FixTime is a BASIC function contained in the DATETIME.BAS file. This file must be loaded as a module and later linked with your main BASIC program.

  FixTime$ lets you accept a time from the user and use that as an argument to PackFilesD, confident that the information is in the correct format.

QUICKPAK PROFESSIONAL ROUTINES

The routines described in this section are from our QuickPak Professional product, and they are included for use by the backup and restore subprograms and also by CWPACK. They are documented here because you are likely to find them useful, especially if you don't already have QuickPak Professional.

This page intentionally left blank.

DCount (function)

- Purpose:

  DCount reports the number of directories that match a given specification.

- Syntax:

```
Count = DCount%(DirSpec$)
```

- Where:

  DirSpec$ holds a DOS directory name specification such as "C:*.*", and Count receives the number of matching directory names.

- Comments:

  Because DCount has been designed as a function it must be declared before it may be used.

  DCount is intended to provide a count of the directories in preparation for using ReadDir to read their names. Where FCount provides a count of file names that match a given specification, DCount instead searches for directory names.

  Most people think of the DOS wild cards (? and *) as being applicable only to file names. However, they are also intended to be used with directory names. For example, to determine the number of directories that are under the root directory of drive C: you would use "C:*.*" as the search specification.

ErrorMsg (function)

- Purpose:

  ErrorMsg returns an appropriate message string given any of the BASIC error numbers that relate to a DOS error.

- Syntax:

```
Message$ = ErrorMsg$(ErrorNumber%)
```

- Where:

  ErrorNumber% is a valid BASIC error number for a DOS operation, and Message$ receives the equivalent message text. For example, given the value 53 ErrorMsg$ will return the string "File not found".

- Comments:

  Because ErrorMsg has been designed as a function it must be declared before it may be used.

  Regardless of how you intend to handle DOS and other errors in your programs, at some point you will probably need to print a message that indicates what went wrong. ErrorMsg provides a simple way to add the standard BASIC error messages without requiring the string memory that would otherwise be taken if you had a series of PRINT or assignment statements.

  The text for each message is kept in a table in the program's code segment, and this table is organized such that it may be easily modified or expanded. See the assembly language source code for details on how to do this if you are so inclined.

Exist (function)

- Purpose:

  Exist will quickly determine the presence of a file.

- Syntax:

```
There = Exist%(FileName$)
```

- Where:

  FileName$ is the file or file specification whose presence is being determined, and There is assigned either to -1 if the file exists, or 0 if it does not.

  The FileName$ parameter may optionally contain a drive letter, a directory path, and either of the DOS wild cards. For example, "B:\STUFF*.BAS" would tell if any BASIC program files are present on drive B in the \STUFF directory.

- Comments

  Because Exist has been designed as a function, it must be declared before it may be used.

  The main purpose of Exist is to prevent the error caused by attempting to open a file for input when it does not exist. Rather than having to set up an ON ERROR trap just prior to each attempt to open a file, Exist will directly tell if the file is present.

FClose (subroutine)

- Purpose:

  FClose closes a file that was previously opened with FOpen.

- Syntax:

```
CALL FClose(Handle%)
```

- Where:

  Handle% is the DOS file handle that was returned when the file was first opened.

- Comments:

  The only error likely to occur when using FClose would be caused by giving an invalid file handle. Errors are detected using the WhichError function described elsewhere in this manual.

  Be careful not to accidentally call FClose with a value of zero, or you will disable the keyboard requiring the PC to be rebooted. FOpen never returns a handle value of zero, but this could happen if you accidentally misspell the handle variable's name.

FCount (function)

- Purpose:

  FCount reports the number of files that match a given specification.

- Syntax:

```
Count = FCount%(FileSpec$)
```

- Where:

  FileSpec$ holds a DOS file name specification such as "C:\BASIC*.BAS", and Count receives the number of matching files.

- Comments:

  Because FCount has been designed as a function it must be declared before it may be used.

  FCount is intended to provide a count of the files in preparation for using ReadFile to read their names.

  You will generally use the DOS wild cards (? and *), to include all of the files in a given directory or in a particular group. FCount also accepts an optional drive and directory as part of the file specification.

FCreate (subroutine)

- Purpose:

  FCreate is used to create a file in preparation for opening and writing to it with the FOpen routine.

- Syntax:

```
CALL FCreate(FileName$)
```

- Where:

  FileName$ is any legal file name such as "ACCOUNTS.DAT" or "C:\CONFIG.BAK"

- Comments:

  FCreate serves the same purpose as BASIC's OPEN FOR OUTPUT command followed immediately by CLOSE. If the file does not exist it will be created, and if it is already present it will be truncated to a length of zero. Programs that use FCreate do not have to use ON ERROR, which results in improved efficiency.

  FCreate will not cause an error if the disk is full, because it does not attempt to write data to the disk—it merely establishes a directory entry for the file. In fact, if a file with the same name exists and also contains data, FCreate will free up the disk space that had been occupied.

  There are two possible causes for an error when using FCreate. Either an invalid file name was given (perhaps a non-existent path or the name contains wild cards or other illegal characters), or the disk's directory is full. For example, a 360K floppy disk can accommodate only 112 entries in its root directory, even if the data area is not filled up.

  Errors may be detected with the WhichError function described elsewhere in this manual.

FGetA (subroutine)

- Purpose:

  FGetA is similar to BASIC's Binary file mode GET command, except it can read any type of data up to 64K in one operation.

- Syntax:

```
CALL FGetA(Handle%, SEG Array(Element), NumBytes%)
```

- Where:

  Handle% is the DOS file handle that was returned by FOpen when the file was opened, and Array(Element) specifies where in the array to begin placing the data as it is read from the file. NumBytes% indicates the number of bytes to read. If the number of bytes is between 32,767 and 65,535 you should use a long integer variable.

- Comments:

  FGetA (the A stands for Array) reads data from the specified file at the location held in the DOS file pointer. The current pointer position is determined by the most recent read or write operation, or by an explicit Seek command.

FLof (function)

- Purpose:

  FLof returns the length of a file that has been opened using FOpen. It is equivalent to BASIC's LOF() function.

- Syntax:

```
Length = FLof&(Handle%)
```

- Where:

  Handle% is the DOS file handle that was returned by FOpen when the file was opened, and Length receives the file's length in bytes. If the handle is invalid, Length instead receives a value of -1 and the WhichError function is set accordingly.

- Comments:

  Because FLof has been designed as a function it must be declared before it may be used.

FOpen (subroutine)

- Purpose:

  FOpen is used to open a file in preparation for reading from or writing to it using the various QuickPak Professional file access routines.

- Syntax:

```
CALL FOpen(FileName$, Handle%)
```

- Where:

  FileName$ is the name of the file that is to be opened, and Handle% is assigned by DOS and returned to you for all subsequent references to the file. If the file does not already exist or any other error occurs, Handle% will be set to -1 to indicate the error.

- Comments:

  You may give either a file name alone, or optionally include a drive, a path, or both.

  It is up to your program to remember the handle that is returned, and use that handle whenever you access the file again later.

  Programs that use FOpen do not have to rely on BASIC's awkward ON ERROR mechanism.

FormatDiskette (function)

- Purpose:

  FormatDiskette lets you add disk formatting capabilities to your programs.

- Syntax:

```
Result = FormatDiskette%(DriveNumber%, Capacity%, SEG BufArray%(1))
```

- Where:

  DriveNumber% refers to a physical drive number with drive A represented as 0, drive B as 1, and so forth.

  The Capacity% argument is given as whole integer values:

  360 = 360K 5.25"
  1200 = 1.2MB 5.25"
  720 = 720K 3.5"
  1440 = 1.44MB 3.5"

  BufArray%() serves as a block of memory that FormatDiskette uses as a work area to hold the disk's FAT (File Allocation Table) as it is being built.

  Result then receives a code that tells if formatting was successful. See the table below for a list of all the possible result codes.

- Comments:

  Because FormatDiskette has been designed as a function it must be declared before it may be used.

  We recommend that you use an integer array as a buffer because it can be dimensioned before formatting the diskette, and then erased afterward. We designed FormatDiskette to require a user-supplied buffer to avoid having it take the necessary memory permanently from your program.

  The table that follows shows how big the buffer must be for each of the supported diskette capacities. Of course, the buffer can be larger than necessary and you can use the largest size only, to avoid having to add extra logic to your programs.

DISK SIZE	BYTES NEEDED	NUMBER OF INTEGER ELEMENTS
360 KB	1060	REDIM BufArray%(1 TO 530)
1.2MB	3644	REDIM BufArray%(1 TO 5822)
720KB	1572	REDIM BufArray%(1 TO 786)
1.44MB	4680	REDIM BufArray%(1 TO 2340)

In this next table, notice that a result code of zero indicates that the diskette was formatted successfully; any other value means there was an error.

ERROR	DESCRIPTION
0	No error
1	Invalid disk parameters
2	Address mark not found
3	Write protect error
4	Requested sector not found
5	Can't locate drive
6	Disk change line is active
7	Invalid capacity specified
8	DMA overrun
9	DMA boundary error
10	Track zero is bad
11	Bad sectors found and marked (not fatal)
12	Media type not found
16	CRC read error
32	Disk controller failure
64	Seek failure
128	Drive not ready

The only error that is not fatal is 11, which means that bad sectors were found but they were marked as being bad and will thus cause no harm. This is the same way the DOS FORMAT program works—as long as track zero is not defective the rest of the disk is still usable.

FormatDiskette can be used to format a disk to a lower capacity than the drive is capable of. For example, you can specify a capacity of 360KB even if the disk drive can handle 1.2MB disks. Likewise, you can specify that a 1.44MB diskette be formatted to only 720KB.

FPutA (subroutine)

- Purpose:

  FPutA is similar to BASIC's Binary file mode PUT command, except it can write any type of data up to 64K in one operation.

- Syntax:

```
CALL FPutA(Handle%, SEG Array(Element), NumBytes%)
```

- Where:

  Handle% is the DOS file handle that was returned by FOpen when the file was opened, and Array(Element) tells where in the array the data being written to the file begins. NumBytes% indicates the number of bytes to write. If the number of bytes is between 32,767 and 65,535 you should use a long integer variable.

- Comments:

  FPutA (the A stands for Array) writes data to the specified file at the location held in the DOS file pointer. The current pointer position is determined by the most recent read or write operation, or by an explicit Seek command.

GetAttr (function)

- Purpose:

  GetAttr examines the directory entry for a specified disk file, and reports the setting of the file's DOS attribute byte.

- Syntax:

```
Attribute = GetAttr%(FileName$)
```

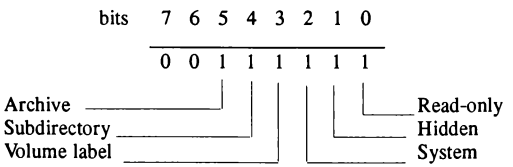
- Where:

  FileName$ is the file being examined, and Attribute is assigned bit-coded with the file's attributes. If the file does not exist or any other error occurs, the attribute value will be returned as -1.

- Comments:

Because GetAttr has been designed as a function it must be declared before it may be used.

The attribute returned by GetAttr is in the form of a single byte, and the placement of the various bits is illustrated in the table below:



You can use BASIC's AND operator to test the individual bits in the attribute byte. For example, to see if a file has been marked as read-only you might do this:

```
IF GetAttr%(FileName$) AND 1 THEN
  PRINT FileName$; " is a read-only file."
END IF
```

We use GetAttr in the CWBackUp subprogram to test each file's Archive bit. This lets CWBackup determine which files have been modified since the last backup, and thus must be backed up again.

GetDisketteType (function)

- Purpose:

  GetDisketteType returns the type of floppy disk drive that is installed.

- Syntax:

```
Result = GetDisketteType%(DriveNumber%)
```

- Where:

  DriveNumber% refers to the physical drive number as recognized by the BIOS. That is, drive A is specified with a value of 0, drive B with a value of 1, and so forth.

  The result returned indicates the type of drive as follows:

  0 = Drive not present or cannot be identified

  1 = 360KB 5.25" 40 tracks

  2 = 1.2MB 5.25" 80 tracks

  3 = 720KB 3.5" 80 tracks

  4 = 1.44MB 3.5" 80 tracks

- Comments:

  Because GetDisketteType has been designed as a function it must be declared before it may be used.

  In most cases you will use GetDisketteType before calling FormatDiskette, to ensure that you specify appropriate parameters. Once the drive type is known you can then proceed to format the diskette.

GetVol (function)

- Purpose:

  GetVol obtains the disk volume label for either a specified drive or the current default drive.

- Syntax:

```
Volume$ = GetVol$(Drive$)
```

- Where:

  Drive$ is either an upper or lower case letter that represents the disk drive to check, or a null string to indicate the current default drive. Volume$ is then assigned the disk's volume label, or a null string if there is no label.

- Comments:

  Because GetVol has been designed as a function it must be declared before it may be used.

  Some programmers like to use a disk's volume label as a way of keeping track of its contents. We use the disk volume label in the CWBackup and CWRestore routines to be sure the correct disk has been inserted during the restore process.

InterruptX (subroutine)

- Purpose:

  The version of InterruptX provided with the Compression Workshop is similar to the routine of the same name that comes with most versions of Microsoft BASIC.

- Syntax:

```
CALL InterruptX(IntNumber%, Registers)
```

- Where:

  IntNumber% is the interrupt to invoke, and Registers is the TYPE variable that holds the register values for the interrupt.

- Comments:

  This version of InterruptX is from our P.D.Q. product, and it differs from the regular BASIC version in that only one instance of the Registers TYPE variable is passed. The same TYPE variable is used for both the incoming and outgoing register values. We made this change to improve the efficiency of the routine.

  Since InterruptX is not a built-in BASIC command, providing it in the same Quick Library with the other Compression Workshop routines eliminates your having to combine the BASIC version with the CWSHOP.QLB file.

MakeDir (subroutine)

- Purpose:

  MakeDir creates a subdirectory in the same way as BASIC's MKDIR command, but without requiring the use of ON ERROR.

- Syntax:

```
CALL MakeDir(DirName$)
```

- Where:

  DirName$ is the name of the directory to create, and may optionally include a drive letter or path.

- Comments:

  The only errors that are likely to happen when calling MakeDir are specifying an invalid drive letter, or a parent directory that does not exist. Errors are detected with the WhichError function described elsewhere in this manual.

MidChar (function)

- Purpose:

  MidChar returns the ASCII value for a single character within a string.

- Syntax:

```
Char = MidChar%(Work$, Position%)
```

- Where:

  Char receives the ASCII value for the specified character, or -1 if Position% is less than 1 or past the end of Work$.

- Comments:

  Because MidChar has been designed as a function it must be declared before it may be used.

  MidChar is substantially faster and more efficient than BASIC's MID$ function. MID$ extracts a copy of the specified portion of the string, and therefore must allocate memory to hold that copy. Since MidChar is an integer function it can operate much more quickly than MID$. Further, integer comparisons are always more efficient than equivalent string operations, which provides even more improvement when you subsequently compare the value returned by MidChar.

PDQTimer (function)

- Purpose:

  PDQTimer returns the number of timer ticks stored in the BIOS data area in low memory.

- Syntax:

```
NumTicks = PDQTimer&
```

- Where:

  NumTicks receives the contents of the four-byte system timer at Hex address 0040:006C in low memory.

- Comments:

  Because PDQTimer has been designed as a function it must be declared before it may be used.

  Unlike BASIC's TIMER function that requires the use of floating point math, PDQTimer returns a long integer result. For programs that otherwise do not need floating point math, using PDQTimer can reduce the .EXE file size by as much as 10K.

PutVol (subroutine)

- Purpose:

  PutVol creates a disk volume label for either a specified drive or the current default drive. If a volume label already exists PutVol will replace it.

- Syntax:

```
CALL PutVol(Drive$, Label$)
```

- Where:

  Drive$ is either an upper or lower case letter that represents the disk drive to access, or a null string to indicate the current default drive. Label$ is then written to the disk's directory as the new volume label.

- Comments:

  Some programmers like to use a disk's volume label as a way of keeping track of its contents. We use the disk volume label in the CWBackup and CWRestore routines to be sure the correct disk has been inserted during the restore process.

ReadDir (subroutine)

- Purpose:

  ReadDir obtains a list of directory names from disk, and loads them into a conventional (not fixed-length) string array in one operation.

- Syntax:

```
CALL ReadDir(BYVAL VARPTR(Array$(0))
```

- Where:

  Array$(0) holds the directory name search specification, and subsequent array elements receive the matching directory names.

- Comments:

  It is essential that enough elements be set aside in the array to hold all of the anticipated directory names. Further, each element must be initialized to a length of at least 12 spaces to hold the name. You should invoke the DCount function to determine how large to dimension the array, and then use a FOR/NEXT loop to initialize each array element before calling ReadDir:

```
Spec$ = "C:\*.*"            'this is the specification
NumDirs = DCount%(Spec$)    'how many names match?
REDIM Array$(0 TO NumDirs) 'create an array to hold them

FOR X = 1 TO NumDirs        'pad each element to 12 spaces
  Array$(X) = SPACE$(12)
NEXT

Array$(0) = Spec$           'show ReadDir what names to load
CALL ReadDir(BYVAL VARPTR(Array$(0))  'read the names

FOR X = 1 TO NumDirs        'display the results
  PRINT Array$(X)
NEXT
```

ReadFile (subroutine)

- Purpose:

  ReadFile obtains a list of file names from disk, and loads them into a conventional (not fixed-length) string array in one operation.

- Syntax

```
CALL ReadFile(BYVAL VARPTR(Array$(0))
```

- Where:

  Array$(0) holds the file name search specification, and subsequent array elements receive the matching file names.

- Comments:

It is essential that enough elements be set aside in the array to hold all of the anticipated file names. Further, each element must be initialized to a length of at least 12 spaces to hold the name. You should invoke the FCount function to determine how large to dimension the array, and then use a FOR/NEXT loop to initialize each array element before calling ReadFile:

```
Spec$ = "C:\BASIC\*.BAS"              'this is the specification
NumFiles = FCount%(Spec$)             'how many names match?
REDIM Array$(0 TO NumFiles)           'create an array to hold them

FOR X = 1 TO NumFiles                 'pad each element to 12 spaces
  Array$(X) = SPACE$(12)
NEXT

Array$(0) = Spec$                     'show ReadFile what names to load
CALL ReadFile(BYVAL VARPTR(Array$(0)) 'read the names

FOR X = 1 TO NumFiles                 'display the results
  PRINT Array$(X)
NEXT
```

SetAttr (subroutine)

- Purpose:

  SetAttr sets a file's attributes entry in the disk directory.

- Syntax:

```
CALL SetAttr(FileName$, Attribute%)
```

- Where:

  FileName$ is the file in question, and Attribute% is bit-coded with the new attributes to set it to.

- Comments:

  Every file has an attribute that is assigned at the time it is created. The attribute information is kept in a disk's directory, along with each file's name, date, time, and size.

  The table below shows some common values for the attribute byte:

1 =	Read-only
2 =	Hidden
32 =	Archive
0 =	Remove all attributes

  We use SetAttr in the CWBackup subprogram to clear each file's Archive bit as it is being backed up. See GetAttr for more information on each of the possible file attributes.

WhichError (function)

- Purpose:

  WhichError reports if an error occurred during the most recent call to a QuickPak Professional DOS routine or function.

- Syntax:

```
Result = WhichError%
IF Result THEN PRINT "Error"; Result; "occurred."
```

- Where:

  Result receives a value that corresponds to a BASIC error number, or zero if no error occurred. For example, calling FOpen with the name of a file that doesn't exist causes WhichError to return a value of 53 ("File not found").

- Comments:

  Because WhichError has been designed as a function it must be declared before it may be used.

  All of the QuickPak Professional DOS routines that are provided in the Compression Workshop can handle and trap DOS critical errors. WhichError is therefore used to prevent fatal errors in the CWBackup and CWRestore routines that would otherwise crash your program if the disks are not ready.

Chapter 3

Utilities

UTILITIES

THE CWPACK AND CWUNPACK UTILITIES

CWPACK and CWUNPACK serve both as demonstrations for using the various Compression Workshop routines, and also as useful utility programs in their own right. These utilities are modeled loosely after the popular PKZIP and PKUNZIP programs, and they let you easily compress, combine, and decompress one or more disk files in a single compressed file.

You specify which files are to be compressed or decompressed and what options you want using command-line parameters. Several of the options require an additional parameter—for example, the /L switch lets you specify a list file which holds the names of the files to be added or removed from the compressed file. Each item in the list file may be either a file name or a legal file specification such as *.BAS, and each item should be on a line by itself.

You may either use a switch alone, or follow it with the information that is expected. In this case, if you enter /L with no file name you will be prompted for it. Otherwise you can specify the name after the /L switch like this:

```
CWPACK FILENAME /L LISTFILE.LST
```

These programs are provided in both BASIC and executable form. We used our P.D.Q. library to create the .EXE versions of these utilities to make them as small as possible. It is not necessary to own P.D.Q. if you want to modify and recompile them. However, BASIC's COMMAND$ function converts all text to upper case, so linking with P.D.Q. is needed to preserve the capitalization of file comments passed on the command line.

Using CWPACK

CWPACK lets you compress any number of input files and combine them into a single compressed file. You can optionally create multiple compressed files to allow fitting many input files onto a series of floppy disks.

CWPACK can also create a self-extracting .EXE program from a compressed file. When run, the program extracts all of the files it contains to the current directory, and optionally runs another program.

The general syntax for using CWPACK is as follows:

```
CWPACK filename [options]
```

The file name must be given first on the command line before any of the optional switches. If no extension is given .CWF is assumed. When adding or updating files in a compressed file, a file specification of "*.*" is assumed unless you specify otherwise. When removing files from a compressed file you must give a file specification. The question mark wildcard character (?) is not recognized by CWPACK except when adding or appending files.

Each of the available option switches is listed below, along with a brief description. Those options that accept a parameter have the parameter shown in brackets. All of the parameters shown in brackets are mandatory, except the keyboard command that can be added to self-extracting compressed files. Again, if a switch that requires a parameter is used but the parameter is not given, you will be prompted for it when the program runs.

/A [filespec] Specifies which files are to be appended to an existing compressed file.

/C [comment] Add a comment to the compressed file. Comments may be added to newly created files only; you cannot add or change the comment in an existing compressed file.

/D [date [, time]] Compress only files that are later than the specified date and time. By default all of the files in the current directory that are newer than the given date will be added, though you can override that with the /S switch (see below). Using /S therefore lets you further discriminate as to which files will be added.

Notice that the time is optional and, if given, must be separated from the date with a comma. If the file time is omitted, midnight is assumed.

/L [filename] Use a response file which contains a list of file names. When /L is used, each file whose name appears in the list is added to or removed from the compressed file. The /L option is mutually exclusive of the /S switch, and any specification given using /S is ignored when /L is used.

/M [program] Convert a compressed file to a self-extracting .EXE program. If /M is used you may also specify the name of a program that is to be run automatically when the self-extracting program finishes. The name of the program is limited to eight characters, and thus cannot include a drive or path specification.

/R [filespec] Remove files from a compressed file. You may either give a DOS file specification, or use /R in conjunction with the /L switch. If you use /L CWPACK will read the names of the files to be removed from the specified list file.

/S [filespec] File specification for adding input files into a compressed file. By default, CWPACK uses *.* when determining which files are to be added. You can override that with the /S switch, either to include only certain files or to specify that the files are located in another directory.

You may use only one file specification at a time with CWPACK. To add, say, all of the files that match *.BAS and also *.MAK you should invoke CWPACK twice—the first time to create the compressed file including *.BAS, and the second time to append to it with *.MAK.

/U Update an existing compressed file from files in the current directory. When /U is given, all of the files that are present in the compressed file *and are also in the current directory* will be updated. If you want to update files individually you must use CWPACK first with /R (remove) and then again with /A (append).

/V View a compressed file's contents. This leaves the compressed file unchanged, and simply displays a list of the files it contains along with their original dates, times, and sizes.

Using CWUNPACK

CWUNPACK extracts files from a compressed file, and is similar in operation to CWPACK. The general syntax for using CWUNPACK is:

```
CWUNPACK filename [options]
```

All of the available option switches that CWUNPACK recognizes are listed below.

/D	Unpack only files that do not exist or are newer than existing files. You may use /D either with or without a list file (see /L below).
/L [filename]	Use a list file containing the names of the files to be unpacked.
/O [path]	Specify a destination path to override the path that may already be present in the list file. This option is for use in conjunction with /L only, and is ignored if /L is not used.
/S [filespec]	Specify which files are to be decompressed.
/T [pathname]	Specify a target directory into which the files will be decompressed. This switch is not meant for use with a list file; that is what /O is for.
/V	View a compressed file's contents. This leaves the compressed file unchanged, and simply displays a list of the files it contains along with their original dates, times, and sizes.

THE INSTALL UTILITY

This program installs applications contained in one or more compressed files (having a .CWF extension), and those files may be installed from any number of distribution diskettes. There are several installation options for added flexibility: you may direct INSTALL to place each .CWF file's contents into the user's default drive and directory, a single drive and/or directory that you specify, or even different directories for each .CWF file. You may also use the .CWF file comment fields to display a brief description of what they contain.

Using Install

Individual .CFW files are selected for installation by the user by pressing the space bar. Each time the spacebar is pressed while a particular .CWF

file name is highlighted, a check mark is turned on or off. Once the desired files are marked for installation the user presses F2 to begin.

If the destination directory does not exist INSTALL will create it. The Tab key toggles between editing the destination directories and selecting files for installation. The Escape key ends the program. These options are shown at the bottom of the main screen, and operating INSTALL should be simple enough for even the most inexperienced user to comprehend.

Setting Up For Installation

You may use any number of distribution diskettes, and each may contain up to 19 .CWF files. However, if you will be installing from more than one disk you must put a file named NUMDISKS.# onto the first distribution disk (the one that has INSTALL.EXE on it and that you label as "Disk 1"). The NUMDISKS extension (shown above as "#") indicates the total number of disks, so INSTALL will know to prompt for additional disks. You can easily create an empty NUMDISKS file from the DOS command line like this:

```
REM > A:NUMDISKS.2
```

Note that DOS won't copy a zero-length file, so this file should be created directly onto the master distribution disk.

Setting Destination Directories

There are several ways to establish default destination directories for the files being installed, and a drive letter may also be included as part of the default directory:

1. If you do nothing at all, the user's current drive and directory are displayed in the "Destination Directory" field for each of the files being installed.

2. You can specify a global default destination directory by placing a file named DEFAULT.DIR in the root directory of the first distribution disk. The contents of this file specifies the default directory to install all of the .CWF files to:

```
COPY CON: A:DEFAULT.DIR <Enter>
C:\PRODUCT<Enter>
<F6> <Enter>
```

3. You can imbed a directory name into a .CWF file's comment field. Any directory names found in a comment field override the directory given in the DEFAULT.DIR file if one was used.

If a .CWF file has a comment field, the comment is displayed to the right of the file name in the selection menu. To imbed a destination directory within a comment, append a CHR$(254) box symbol followed by the directory name:

```
This contains the main program ■ C:\DIRNAME
```

Here, the text "This contains the main program" is displayed on the screen, so the user will know the purpose of this particular .CWF file. The drive and directory C:\DIRNAME is used as the default destination for installation of this file only. If you prefer, you can specify a default directory without adding a displayable comment by using only a box and the directory name for the .CWF file's comment:

```
■\DIRNAME
```

or

```
■D:\DIRNAME
```

Regardless of whether or not you assign a default destination directory, the user can override the default and specify a different drive and directory.

You can enter a box character in most editors by pressing and holding the Alt key, and then typing the digits 2-5-4 in succession on the numeric key pad. After you have pressed the third digit, release the Alt key and the box symbol will appear.

Selecting Files for Installation

In some cases your users may not want or need to install all of the files on the distribution disks. For example, we provide the assembler source code with our products, but many people aren't interested in assembly language and may prefer not to waste their disk space with those files.

To have a file selected for installation automatically, append a CHR$(251) check mark (√) to the end of the comment string, but before the optional destination path:

```
This contains the main program √■C:\DIRNAME
```

or

```
This is a comment with no path√
```

or

```
√■C:\PATHNAME
```

Files whose comments have no trailing check mark will not be selected, though the user may of course select those files manually.

The comment portion (before the box and optional check mark) can be up to 36 characters long, and the directory name portion of the comment (after the box) can hold as many as 25 characters.

You can also tell INSTALL to run a program automatically by storing its name in a file named PROGRAM.RUN, and placing that in the root directory of your first distribution disk. The name of the program must be no longer than 14 characters, since INSTALL uses the QuickPak Professional StuffBuf routine to place the name (plus Enter) into the keyboard buffer.

Installing to Nested Directories

If you are installing to nested directories you must specify that the higher directory levels be installed first. INSTALL will create the directories if they are not already present, but it must do this from the top down. For example, if you have main files that are to go in a directory named \APPMAIN and ancillary files that are to go in \APPMAIN\SUBDIR, INSTALL must create \APPMAIN before it can create \APPMAIN\SUBDIR.

As long as the .CWF files on your distribution disk or disks are in the correct order this will not be a problem. However, it is possible for a user to screw this up by changing the directory names during installation. There is little that can be done about this problem, and if INSTALL is unable to create a directory is simply ends with an error message. Likewise, if you include a DEFAULT.DIR file it should contain only a root-level directory unless you are certain that the upper levels already exist.

Composite Monitors

INSTALL detects whether a color or monochrome adapter is present, and adjusts its colors accordingly. Since some die-hards are still using computers with a composite display (CGA adapter connected to a monochrome monitor), you can tell your users to use /B on the command

line if the display is unreadable. Many programs use /B for this purpose, including the Microsoft BASIC editors.

THE BACKUP AND RESTORE SUBROUTINES

BACKUP.BAS holds both the backup and restore subprograms, and these are named CWBackup and CWRestore respectively. DEMOBACK.BAS is a full-featured backup program that also serves as a demonstration, and DEMOREST.BAS shows how to use the restore subprogram. This section provides an overview of the backup and restore subprograms and also detailed syntax descriptions. After that is a discussion of modifications you may want to make to these routines.

Most modern computers support what is known as the ChangeLine switch, to let a program know that a disk has been removed from the drive since the last access. When backing up and restoring on these computers the routines can determine when the disk is changed automatically and proceed accordingly. But with older computers the user must press a key to indicate that the next disk has been inserted. The prompt messages displayed by CWBackup and CWRestore are discussed later in the section *Modifying CWBackup and CWRestore*.

Using CWBackup

CWBackup accepts a drive letter and starting path, to tell it what files are to be backed up. You will also tell CWBackup whether it is to include the subdirectories beneath the initial starting directory, and the drive letter to store the backup files onto. If one or more of the target diskettes are not formatted CWBackup will format them automatically, before attempting to backup files to them.

Only files whose Archive attribute bit is set are backed up, and as each file is backed up the Archive attribute is cleared. This is how most commercial backup utilities work. Comments in the BASIC source code show how to change CWBackup to not test or clear the Archive attribute if you prefer.

The calling syntax for CWBackup is as follows:

```
CALL CWBackup(FileSpec$, Dest$, Recurse%, ErrCode%)
```

Here, FileSpec$ indicates which files are to be backed up. To back up an entire hard disk you would specify all of the files starting in the root directory. For example:

```
FileSpec$ = "C:\*.*"
```

Dest$ specifies the destination drive letter, and only the first character is used.

The Recurse% parameter is set to either -1 to include all subdirectories under the starting directory, or zero to backup only the stated directory. If you enter /S on the DEMOBACK.BAS command line it will set the Recurse% parameter to -1 to include subdirectories in the backup set.

ErrorCode% is used for two purposes: to specify a timeout value when calling CWBackup, and also to return an error code upon completion. This was done to reduce the number of parameters needed. When calling CWBackup you should set ErrCode% to the number of seconds it is to allow for changing disks. The demonstration programs use a value of 20 for this, which means the user has up to 20 seconds to change disks each time a new disk is prompted for. If a disk isn't changed within that time, CWBackup will return with ErrCode% set to 255. Any other non-zero ErrCode% value indicates one of the Compression Workshop's standard error codes. These are described in the section entitled *Error Codes*.

The disk volume label on each of the backup disks is set to a unique value by CWBackup. This volume label is then used later by CWRestore to ensure the files are restored in the correct order.

Using CWRestore

CWRestore accepts both a source and destination file specification, to tell it which files to restore and where to place them. Like CWBackup, CWRestore lets you restore either all directories or only the top level directory—even if all levels were backed up originally.

You can restore the files to either their original drive and directory, or to any arbitrary drive or directory. If the destination directory does not yet exist CWRestore will create it automatically. The volume label for each disk in the backup set is examined by CWRestore to ensure the files are handled in the correct order.

The calling syntax for CWRestore is as follows:

```
CALL CWRestore(FileSpec$, Dest$, Recurse%, ErrCode%)
```

FileSpec$ indicates which files are to be restored, and is specified using a full drive and path as follows:

```
FileSpec$ = "d:\path\spec"
```

Here, the "d:" portion of FileSpec$ is the letter of the floppy disk *from which you are restoring*, and the "spec" portion specifies which files are to be restored (for example, *.BAS). The "\path\" portion is optional, and it indicates the path *from which the files being restored originally came*. This lets you indicate which files are to be restored, when the destination directory is different from the directory in which the files resided originally. To restore all of the files on drive A: that have a .BAS extension and were originally backed up from a directory named \BASIC you will use this:

```
FileSpec$ = "A:\BASIC\*.BAS"
```

Again, the drive letter A: tells CWRestore where the backed up files now reside, and the path \BASIC tells CWRestore which files are to be restored. This lets you distinguish the files you want to restore, in case other files in the backup set also happen to have a .BAS extension.

Dest$ specifies both the destination drive letter, and also an optional destination path to let you restore to a directory that is different from the one in which the files originally came. If no path is given then the files are restored to the same directories from which they were backed up.

As with CWBackup, the Recurse% parameter is set to either -1 to restore all subdirectories under the starting backed up directory, or zero to restore only the top-level directory. Enter /S on the DEMOREST.BAS command line to set the Recurse% parameter to -1 telling it to include subdirectories.

Also like CWBackup, ErrorCode% specifies both the timeout value when calling CWRestore and an error code upon return. When calling CWRestore you set ErrCode% to the number of seconds to allow for changing disks. If a disk is not changed within that time CWRestore will return with ErrCode% set to 255. Any other non-zero ErrCode% value indicates one of the Compression Workshop error codes.

Compiling BACKUP.BAS

You must compile BACKUP.BAS using the /ah (huge array) option switch, or the CWBackup routine will be limited to 512 files in a single backup set. It is not necessary to compile your main program or other modules using /ah, but of course you may if they also require huge array support.

Modifying CWBackup and CWRestore

Both CWBackup and CWRestore need to prompt the user when a new disk is to be inserted. But in the context of add-on subroutines you may well prefer to implement your own user interface. As provided, these routines simply print their messages at the current cursor location. If you wish to change the way messages are displayed, search BACKUP.BAS for the string "Your user interface here" in the CWBackup, CWRestore, ErrorHandler, and InsertDisk subprograms.

Another change you might want to make is not have CWBackup test for and clear each file's archive bit during backup. To do this search BACKUP.BAS for the string "REMark" and follow the instructions shown.

UTILITIES

Chapter 4

Technical Details

TECHNICAL DETAILS

TECHNICAL DETAILS

This library is based on a 9-13 bit implementation of the LZW compression algorithm. To speed up table searching, open address double hashing is used with a 9,029-element hash table. The performance is very effective overall, but is directly related to the type of data being compressed. The sections that follow describe how the LZW algorithm operates, and also how the Compression Workshop file headers are structured.

LZW Compression Overview

The LZW algorithm is a table-based compression scheme. A table is built at runtime, and this table is based upon the input data. The output consists of addresses in that table.

The data is read in a byte at a time. A code and suffix pair is created consisting of the "hold" value and the input byte. The table is searched, and if the pair isn't found it is added to the table and the hold value is output. If it is found, the address where it was found becomes the new hold value.

Consider the following input stream (the spaces are shown for clarity, and are not part of the actual input data):

```
A  B  C  A  B  C  A  B  C  A  B  C  A  B  C
65 66 67 65 66 67 65 66 67 65 66 67 65 66 67
```

T-loc	code	suf	hold	O/P	
256	65	66	66	65	1) The first two bytes are a special case. They are assigned to the table automatically. Then the first byte is output and the second becomes the hold value.
257	66	67	67	66	2) The next byte is read in (67) and the table is searched for the code/suffix pair we created from the hold value and the input byte (66,67). Since it is not found it's added to the table, the hold value is output, and the input byte beomes the new hold value.
258	67	65	65	67	3) The next byte is read in (65) and the table is searched for the code/suffix (67,65). Since it is not found it is added to the table, the hold value is output, and the input byte becomes the new hold value.
			256		4) The next byte is read in (66) and the table is searched for the code/suffix (65,66). Since it does exist, the address where it was found becomes the new hold value.
259	256	67	67	256	5) The next byte is read in (67) and the table is searched for the code/suffix (256,67). Since it is not found, it is added to the table, the hold value is output, and the input byte again becomes the new hold value

continued...

T-loc	code	suf	hold	O/P	
			258		6) The next byte is read in (65) and the table searched for the code/suffix (67,65). Since it does exist, the address where it was found becomes the new hold value.
260	258	66	66	258	7) The next byte is read in (66) and the table is searched for the code/suffix (258,66). Since it is not found it is added to the table, the hold value is output, and the input byte becomes the new hold value.
			257		8) The next byte is read in (67) and the table is searched for the code/suffix (66,67). Since it exists the address where it was found becomes the new hold value.
261	257	65	65	257	9) The next byte is read in (65) and the table searched for the code/suffix (257,65.) Since it is not found it is added to the table, the old hold value is output, and the input byte becomes the new hold value.
			256		10) The next byte is read in (66) and the table searched for the code/suffix (65,66). Since it exists, the address where it was found then becomes the new hold value.
			259		11) The next byte is read in (67) and the table searched for the code/suffix (256,67). Since it exists, the address where it was found becomes the new hold value.
262	259	65	65	259	12) The next byte is read in (65) and the table is searched for the code/suffix (259,65). Since it is not found it is added to the table, the old hold value is output and the input byte becomes the new hold value.
			256		13) The next byte is read in (66) and the table is searched for the code/suffix (65,66). Since it exists, the address where it was found becomes the new hold value.
			259		14) The next byte is read in (67) and the table is searched for the code/suffix (256,67). Since it exists, the address where it was found becomes the new hold value.
				259	15) Once the input stream is exhausted, the last hold value is output.

The above input stream is 120 bits (15 bytes) and the resulting output is 72 bits (8 codes at 9 bits per code), for a net saving of 6 bytes. The output codes are shifted and packed according to the size dictated by the current maximum table address. The Compression Workshop uses 9-13 bits with a maximum table address of 8,191.

As the amount of input increases, the odds of finding a match are improved. And the higher codes generally will represent longer "sub-strings" thus improving compression.

Obviously, a sequential table search would be very slow. Therefore, to speed table searching the Compression Workshop uses open address double hashing with a hash table size of 9,029 bytes.

LZW Decompression Overview

When a code is read in it is automatically placed into the table as the code portion of the next available table location. The suffix portion is filled when the next input code is processed. If the input code is less than 256, then it is directly output and used as the suffix at T-loc minus 1. If the code is 256 or more, the suffix value at the corresponding table location is placed into a LIFO (Last In First Out) stack.

If the code at that table location is less than 256, it is placed into the LIFO stack and used as the suffix at T-loc minus 1. If the code is 256 or more, the suffix value at the corresponding table location is placed into a LIFO stack, and so forth until a code value of less than 256 is found. Then the LIFO stack is output.

Consider the codes output from the above example as input:

```
65 66 67 256 258 257 259 259
```

T-loc	code	suf	O/P	
256	65	66	65	1) The first code is read in and placed in the table. Since the first code is always less than 256 it is output.
257	66	67	66	2) The next code is read in and placed in the table. Since it is less than 256 it is output and placed as the suffix at T-loc minus 1.
258	67	65	67	3) The next code is read in and placed in the table. Since it is less than 256 it is output and placed as the suffix at T-loc minus 1.
259	256	67	65 66	4) The next code is read in and placed in the table. Since it is 256, the suffix value at T-loc 256 (66) is put into the LIFO stack. The code at T-loc 256 (65) is less than 256 so it is placed into the LIFO stack and used as the suffix at T-loc minus 1. The LIFO stack (66,65) is output.
260	258	66	67 65	5) The next code is read in and placed in the table. Since it is 258, the suffix value at T-loc 258 (65) is put into the LIFO stack. The code at T-loc 258 (67) is less than 256, so it is placed into the LIFO stack and used as the suffix at T-loc minus 1. The LIFO stack (65,67) is output.
261	257	65	66 67	6) The next code is read in and placed in the table. Since it is 257, the suffix value at T-loc 257 (67) is put into the LIFO stack. The code at T-loc 257 (66) is less than 256, so it is placed into the LIFO stack and used as the suffix at T-loc minus 1. The LIFO stack (67,66) is output.

continued...

262	259	65	65 66 67	7) The next code is read in and placed in the table. Since it is 259 the suffix value at T-loc 259 (67) is put into the LIFO stack. The code at T-loc 257 is 256, so the suffix value at T-loc 256 (66) is put into the LIFO stack. The code at T-loc 256 (65) is less than 256, so it is placed into the LIFO stack and used as the suffix at T-loc minus 1. The LIFO stack (67,66,65) is output.
263	259		65 66 67	8) The next code is read in and placed in the table. Since it is 259 the suffix value at T-loc 259 (67) is put into the LIFO stack. The code at T-loc 257 is 256, so the suffix value at T-loc 256 (66) is put into the LIFO stack. The code at T-loc 256 (65) is less than 256, so it is placed into the LIFO stack and used as the suffix at T-loc minus 1. The LIFO stack (67, 66, 65) is output. Since that's the last I/P code we're done.

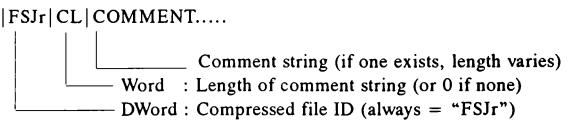
The table created is an exact duplicate of the compression table, but with one extra code.

Compression Workshop File Structures

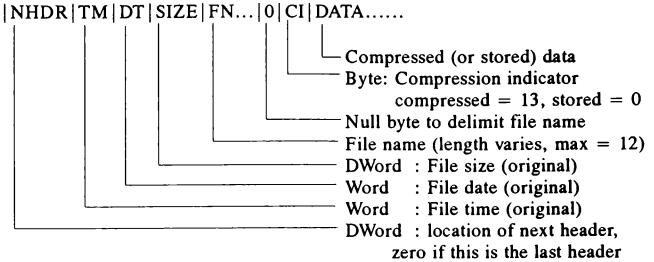
The following details the file structures used by the various Compression Workshop routines.

File Compression Routines

- Main Header:



- File Headers:



Array Compression Routines

- Main Header:

|FSJr|

└DWord : Compressed file ID (always = "FSJr")

- Array Headers: