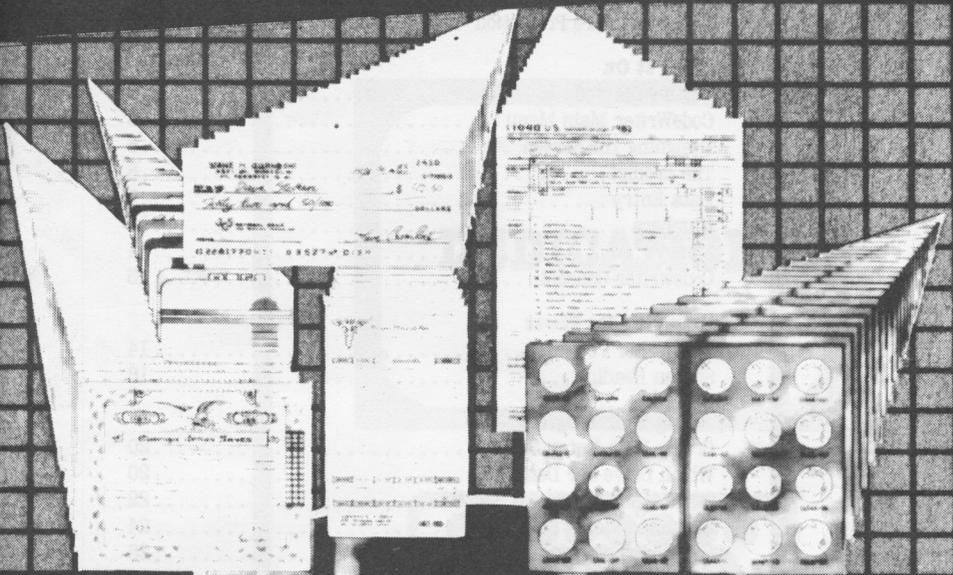


**YOUR OWN PROGRAM**  
— the first time you try —



**DYNATECH**  
SOFTWARE SYSTEMS

# Home FileWriter™

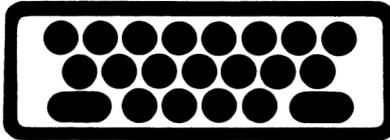
Now your personal computer becomes your **PERSONAL PROGRAMMER!**  
Home FileWriter lets you create **YOUR OWN PROGRAMS ON YOUR OWN DISKS.**

No tricky computer jargon to learn. Just draw your screens **EXACTLY**  
the way you need them — for checks, credit cards, medical  
records, tax information, team sports records, church, **ANYTHING!**  
Needs: Commodore 64, one 5.25" disk drive, printer optional. Item #1-100

# TABLE OF CONTENTS

<b>Introducing Your CodeWriter</b>	
Read This First . . . . .	2
That's Enough Theory . . . . .	2
About Software Protection . . . . .	3
<b>Turning It On</b>	
Getting Started . . . . .	4
CodeWriter Main Menu . . . . .	5
Designing The Screen . . . . .	6
The Prompt . . . . .	8
Data Entry . . . . .	9
Numbers . . . . .	10
Money . . . . .	11
Codewriter Concept . . . . .	13
<b>Multiplying Its Power</b>	
CodeWriter Power . . . . .	14
Screen Reading . . . . .	16
<b>Creating Your Program</b>	
Create Data Entry System . . . . .	20
Which Drive For Data? . . . . .	20
The Grand Total Fields . . . . .	22
Computed Fields . . . . .	24
CodeWriter Concept . . . . .	25
Self-Referencing Fields . . . . .	26
How Many Records Needed? . . . . .	27
The Key Field . . . . .	28
Keeping Out Garbage . . . . .	30
Program Design Creativity . . . . .	32
<b>Running Your Program</b>	
The Maiden Voyage . . . . .	34
File Preparation . . . . .	35
Enter Data . . . . .	36
Update Data . . . . .	36
Look Up Record . . . . .	38
Search Records . . . . .	38
Delete Record . . . . .	40
Verify Grand Totals . . . . .	40
Exit . . . . .	40
A Final Word . . . . .	40
Appendix A . . . . .	41

# DEDICATION



The CODEWRITER Program Series, developed over the last two years, owes its existence to the following—who were never afraid to walk in the dark:

Fred Allen  
Larry Coke  
Dan Kritchevski  
Paul Green  
Warren Shore  
Jay Stein  
Tony Thorne, MBE

Special Commodore CBM 64 and Atari conversion work  
Charles and Carol Butler



## **READ THIS FIRST**

**WATCH FOR THIS SYMBOL.** This manual was written to be used with several microcomputer systems. There will be a number of places where the instructions in the manual won't be right for YOUR computer. Whenever there's a chance of this, you'll see a large \* symbol in the margin. This means you might need to refer to your USER NOTES CARD (supplied with your CODEWRITER SYSTEM). The card should clear things up.

(By the way, your Home FileWriter is part of a family of software called CODEWRITER. All references in this manual will describe "CODEWRITER" or "your CODEWRITER SYSTEM". Your Home FileWriter is the starter system in the series.)

This manual was written to be as unique and useful as the CODEWRITER SYSTEM itself. That is a tall order. Please believe us when we say that everyone at Dynatech Microsoftware worldwide has worked to make CODEWRITER among the most valuable programs you can own.

There has been a great deal written about "Program Generation", both good and bad, pro and con. Let's begin with a simple statement about why THE CODEWRITER SYSTEM was developed over the last two years (since 1980):

There is only one reason for computers. To allow people to control the information around them. But computers deal in code while people deal in ideas. As personal computers become available to more and more people, "programming" in arcane and unforgiving code gets in the way more and more. The same people who understand the information a computer holds must be able to control that information directly. In this way, ideas can dominate rather than hardware.

THE CODEWRITER SYSTEM allows the non-programmer with an idea for ordering information to see that idea take shape on a personal computer. If the idea has elements which can be put on a screen or written on paper, CODEWRITER will structure those elements so a computer can accept them, retrieve them, re-order them and create a pattern for understanding.

Of course, none of this is magic. CODEWRITER only substitutes "programming" with "Program Design". But the difference is critical. The programmer has two problems at all times; One is the idea at hand and the second is the job of reducing that idea to a "language" vastly more primitive than human thought.

While Program Design is a great deal easier, it is hardly trivial. The simplicity of CODEWRITER Program Design comes from dealing with a problem directly in the designer's own language.

### **THAT'S ENOUGH THEORY**

CODEWRITER will create all the computer code needed to get a program up and running on your computer. Once you're done "designing", you'll see the code written out on your computer screen as it is automatically recorded onto your disk.

For the most part, THE CODEWRITER SYSTEM is "self documented"—that is most of what you need to know to design a program is written on the screen for you and will re-appear each time you use the system.

But use this manual anyway! There is very little more "theory" inside. We have designed an example program—complete with every single keystroke needed to create that program using only human language.

Naturally, our example program is unlikely to do exactly the things you bought CODEWRITER to do. For now, that's not important. The example will show a great number (but not all) of the features of the CODEWRITER system.

More importantly, the example program will show how these features work together to solve a complex problem one step at a time.

Put the manual in front of your computer where you can read it comfortably. Turn your computer on and begin a process you'll not soon forget.

You're about to increase your dominion over the single most stimulating invention of the twentieth century. Have fun. We envy you and wish you well.

### **A WORD ABOUT SOFTWARE PROTECTION**

At Dynatech Microsoftware we have some very definite ideas about protecting software. We feel that both the software developer and the software customer have rights which must be protected. The developer must be protected from "unauthorized use" of his work. After all, if the market place does not reward the developer for his work the work will not be produced, not be supported, and not be improved.

But workable software protection cannot exclude the customer's rights. The paying customer makes all new software possible. Thus, the customer should be able to use the software freely and with confidence. A 'back-up' copy of your Commodore 64 CODEWRITER disk is available at a small cost (See the coupon included with your system). Also, a free one year guaranty is part of your system cost. If your CODEWRITER disk fails to operate for any reason during this period, we'll replace it free. Once your purchase is registered, you'll be notified of our toll-free help line for any problems you might have with your CODEWRITER system.

In addition, whenever possible we'll add new features (or improve existing features) to the CODEWRITER system at no increase in retail price. We'll be glad to UPGRADE your existing system free if you'll simply send your disks in. Watch our ads for these changes—or simply call to make sure your version is the latest.

Please enjoy your CODEWRITER system. We developed it to be the best.



## GETTING STARTED

Before you begin working with your Commodore CBM 64 CODEWRITER system, make sure you have the following:

A Commodore CBM 64 microcomputer

A 1541 Commodore disk drive

(In some cases a modified 1540 drive will do—see your dealer. A 4040 dual disk drive will also work, though this edition of CODEWRITER 64 will use only drive 0.)

A blank, formatted diskette.

A Commodore compatible printer is optional.

(SPECIAL NOTE: Some commodore compatible options like Skyles' VicTree will interfere with CODEWRITER operations. Please remove them before you begin. If your system works erratically, check for these optional items before continuing.)

Insert your CODEWRITER disk (Disk 1 side up) in your disk drive. Press the SHIFT and Commodore symbol key (lower left on keyboard) TOGETHER. This will activate the UPPER/lower case mode in which CODEWRITER operates. Next, type the following EXACTLY:

load "newmenu",8

After the program loads, your cursor (flashing square) will return along with the 'READY' message. Type run. After a short delay, the following screen should appear:

### Codewriter System Main Menu

d-create date entry system  
f-format a disk  
s-set display colors  
x-exit to basic

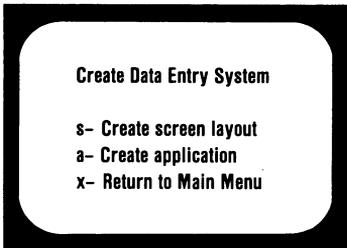
If you have not already formatted a disk, do so now.

Press 'f' and then RETURN

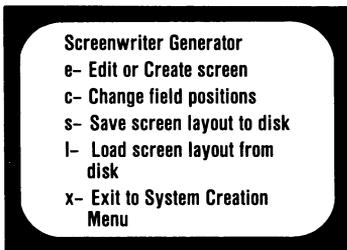
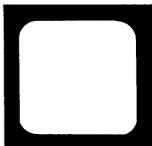
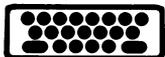
Pressing 'f' will display a warning that formatting ERASES all information on a disk. REMOVE the CODEWRITER disk and insert your own disk. Type 'y' and press RETURN. You'll be asked to "name" your work disk using up to 16 characters and press RETURN. Formatting usually takes between 90 seconds and 2 minutes. You'll then be asked if you need another disk 'formatted'. Answer 'n' and press RETURN. You should now see the CODEWRITER System Main Menu again.

## CODEWRITER SYSTEM MAIN MENU

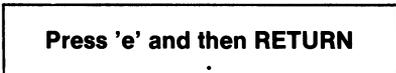
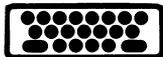
Once back at the main menu with a formatted disk, we're ready to create a data entry system. Press 'd' from the menu selection and then RETURN.



Each CODEWRITER data entry program begins with a SCREEN LAYOUT. This is simply a form created on the screen which shows what kind of information (data) the program operator is to enter and how much space is allowed to do so. CODEWRITER makes this process as easy as possible.



We'll look at all the options on this menu before we're done, but for now the task is to CREATE A SCREEN so:



You should be looking at a screen full of instructions on how to accomplish writing a screen. This is for future reference in using CODEWRITER. For now the screen instructions might make the job look more complex than it really is. Let's examine the instructions to sort things out:

CODEWRITER allows you to type anywhere on the screen to create the entry form you want. The screen instructions will show which keys on your computer allow you to move the cursor around the screen.

Your CODEWRITER 64 system allows you to make printed copies of your screen designs. You must have a Commodore (or compatible) printer connected to your system to do this. You'll see a line at the bottom of your work screen which reads:



Press f1 to read screen f3 hardcopy

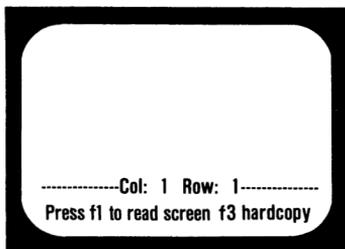
The 'hardcopy' refers to printed copy. Simply press the f3 function key on your Commodore 64 to send an image of your screen design to your printer.

**ONLY ONE LIMITATION**—You may NOT use column 40 of your screen as part of your screen design. The cursor position indicator (second line from the bottom) will let you know when you're in this column. Put NO screen information there.

That's really all there is to writing on the CODEWRITER screen. We will cover all of the information on the current screen as we proceed with the example program. For now, just remember the instruction screen is there to help you. Press RETURN. You'll see another instruction screen. This, too, will be covered in our example program. Again, remember the screens are there and press RETURN.

### DESIGNING THE PROGRAM SCREEN

You should now be looking at an almost blank screen. the CODEWRITER screen is 40 columns across and 22 rows top to bottom. You should see the cursor at the upper left and two lines at the bottom of the screen:



The Col/Row line will TRACK the cursor position on the screen. Try using the cursor keys we described earlier. Watch the numbers on the Col/Row line change as the cursor moves. This CODEWRITER feature helps in counting positions when designing your screen and is very valuable when you're trying to copy an existing form to the screen for use in a program.

Before we begin our example program, we need to understand a few terms about the way computers handle information. The terms are FILE, RECORD, and FIELD:

**FILE**—A FILE is a collection of information on a single subject. Thus a receivable file is a collection of information on who owes money to a particular company. A stamp collection file contains all the information about a certain stamp collection, etc.

**RECORD**—As we get more specific, we use the term RECORD. Thus, within a stamp collection FILE, information about a certain stamp would appear in a RECORD for that stamp. Within a receivables FILE we would find RECORDS of the individual companies or people who owe money.

**FIELD**—The FIELD is the most specific information. Within the stamp collection FILE, the RECORDS for individual stamps would contain FIELDS of information like; the color of the stamp, the country of origin, etc.

Don't be discouraged if everything you read is not clear the first time through. We have tried to keep computer jargon to a minimum in this manual, but a little is bound to creep in. If you work through the example program, things will begin to come together.

Your CODEWRITER form screen is a remarkably flexible tool. Getting information into your program, in the order you want and the language clearest to you should be easy. Don't be afraid to experiment. You may use as many as 100 fields on a screen. Just type anywhere (except col. 40)—YOU CAN'T HURT CODEWRITER OR YOUR COMPUTER FROM THE KEYBOARD.

### A SALES/INVOICE PROGRAM

Our example program is intended to keep track of sales. We chose this program idea because it gives a good indication of what the CODEWRITER system will do. To use our new vocabulary, we wish to build and keep track of a FILE of sales over a certain period of time. Each sale will be entered to a screen RECORD known, naturally enough, as an invoice: Each invoice will contain FIELDS to put the most specific information like; customer name, item purchased, date, price paid, etc.

We should give our invoice form some kind of heading or label to show what its use will be. The example in the shaded box below uses an up arrow ( ↑ ) as a SPECIAL MARKER on either side. Your computer may use another character. Check the screen instructions.

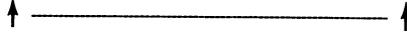


↑ ABC COMPANY SALES INVOICE ↑

What you have typed is known as a LABEL to the CODEWRITER system. A LABEL is something written which is NOT associated with information to be entered. Things like our title (just entered), copyright information on the screen, instructions for the program operator, dotted lines and the like are all LABELS to the system because they DON'T ASK ANYONE TO REACT BY ENTERING INFORMATION.



As you see our label example has the up arrow ( ↑ ) symbol on either side. This identifies screen information as a LABEL. The up arrow key on your Commodore 64 is the second key from the right on the second row from the top. Even if screen material is simply a line as below:



be sure to use UP ARROWS on BOTH SIDES.

By the way, don't be concerned if the invoice label you just typed is not centered exactly as you wish. We'll take care of things like that later.

### THE PROMPT

Now for the part of our sales invoice screen that IS concerned with information handling. Let's add some customer information PROMPTS to our form. A PROMPT asks for INFORMATION TO BE ENTERED. A PROMPT is always followed by at least one dot ( . ) or a dotted line ( . . . . . ) to indicate HOW MUCH SPACE is available to enter the information requested by the PROMPT.

Thus we could add customer information PROMPTS to our sales invoice screen and it would look as follows: (Don't worry about typing exactly.)



↑ ABC Company Sales Invoice ↑

Customer name . . . . .

Street address . . . . .

City . . . . .

There are several things we should notice about PROMPTS. As you can see, there are no UP ARROWS. For the CODEWRITER system to recognize your PROMPTS as the requests for information they are, never use UP ARROWS. Also while each PROMPT must have a dot ( . ) or dotted line ( . . . . . ) following it, the dots DON'T have to come IMMEDIATELY AFTER the letters or numbers in the PROMPT.

Look at the PROMPT 'Customer name'. After the final 'e' in 'name', there are TWO SPACES BEFORE the line of dots begins. This allows you to create screen forms which are easier to read because the PROMPT for information needn't bump right against the information itself.

In any of your screen designs, the number of DOTS which follow the PROMPT determines HOW MANY characters of information (letters, numbers, symbols or spaces) may be entered to answer that prompt.

## CODEWRITER CONCEPT

**THE PROMPT**—A PROMPT is a request for information by the program designer. It is always followed by a dot or dots to indicate length of entry. It NEVER contains up arrows. The CODEWRITER system will search for the FIRST DOT following a PROMPT and store the information which follows as the response to the PROMPT. The total number of dots following a PROMPT should never be more than 38 when using your 40 column computer.

There is one more tip concerning PROMPTS: Never put a dot (.) into the PROMPT itself. This can happen where a PROMPT involves an abbreviation as in—

Max. amount needed? (y or n)

This is simply a PROMPT asking for a yes or no (one letter) response to the question 'Maximum amount needed?' Can you see what's wrong? The CODEWRITER system will see the dot after 'Max' and consider 'Max' alone to be the PROMPT with a one-dot response (Max.) CODEWRITER would then read further (from 'amount needed', etc.) and consider this to be a second PROMPT.

Abbreviation is O.K. Simply leave out the period as in—

Max amount needed? (y or n)

This will work fine.

### DATE ENTRY

The CODEWRITER system handles dates as a special kind of response to a PROMPT. You may use either the American date format or the European and CODEWRITER will automatically write code to check for the appropriate format and a valid date entry, i.e. no July 40 or February 29 (when not a leap year), Formats are as follows: To express the 15th day of June, 1983

American  
06/15/83

European  
15/06/83

For now simply enter the empty date format. CODEWRITER will ask for the American/European choice later.

../././..

Added to our current screen this would be:

↑ ABC Company Sales Invoice ↑

Customer name ..... Date ../././..

Street address .....

City .....

Add the date PROMPT to your screen. Whenever you wish date information to be entered into your CODEWRITER created programs, use the .././.. format. The PROMPT may be whatever you wish as:

Order Date     .././..  
Member since   .././..  
Date to Close   .././..    etc.

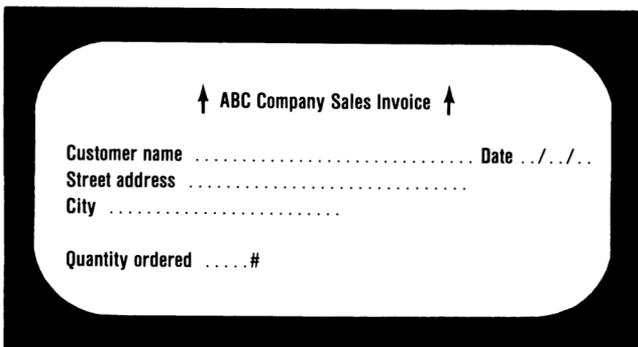
Only the actual date entry format need be the same. When the operator of your program enters a date, it will be as 02/05/81— (You may type SPACES instead of leading zeros.) Later on in program development you will be offered the choice by CODEWRITER as whether you wish American or European format date handling.

### NUMBERS

Up to now the information required by the PROMPTS on our sales invoice screen has been what your computer considers ALPHANUMERICs—jargon meaning IDEAS expressed in letters and numbers. For example a name, an address, and a date are all ALPHANUMERICs because of two things; They can be expressed in letters and numbers AND they are NOT USUALLY part of any CALCULATION—you don't add, subtract, multiply or divide them even though they MAY include numbers.

NUMERICs, to your computer, are different. In the CODEWRITER system NUMERICs have two meanings of their own; They involve NUMBERS ONLY, never letters, and they can be included in CALCULATIONS. As we promised to avoid jargon, let us begin here to refer to NUMERICs as simply numbers.

As we add a new line to our sales invoice screen you'll notice a change:



↑ ABC Company Sales Invoice ↑

Customer name ..... Date .././..  
Street address .....  
City .....  
Quantity ordered .....#

See the new symbol? After the 5 dots following 'Quantity ordered', we've added the # sign. This does two things; The # sign takes the place of a dot, making the space for information total 6, and the # sign tells the CODEWRITER system that the information to be entered will be NUMBERS and ONLY NUMBERS. Thus the numbers may be part of a calculation—if the program designer wishes.

By now your program screen should look like the one above and include the new NUMBERS field for 'Quantity ordered'. Let's add another field to the screen.



```
      ↑ ABC Company Sales Invoice ↑  
  
Customer name ..... Date .././..  
Street address .....  
City .....  
  
Quantity ordered .....# ITEM .....
```

As you see, we've added another ALPHANUMERIC field called 'ITEM' and given it 16 spaces for operator entry. Again, we didn't need any symbol after the row of dots. Add the new field yourself.

Now it's time for a little 'housekeeping'. As we look at the latest line on our sales invoice screen, it looks as though space is running out too soon. Most invoice forms allow for 'Quantity', 'Item', 'price each', and 'total' all on a SINGLE LINE. The CODEWRITER system allows ANYTHING on your form to be retyped as often as you like until it's just as you wish. Why not take advantage?



```
      ↑ ABC Company Sales Invoice ↑  
  
Customer name ..... Date .././..  
Street address .....  
City .....  
  
Quan .....# ITEM .....
```

There. We've abbreviated 'Quantity ordered' to 'Quan' (no period) and added space to the 'ITEM' PROMPT, allowing for a better description of ITEM.

### **MONEY**

The last type of PROMPT field CODEWRITER offers is for MONEY. This field type simply stores numbers for all DECIMAL TYPE CURRENCIES for a maximum of 2 places to the right of the decimal point. The CODEWRITER program designer adds the \$ sign (the meaning here being 'money' rather than the American dollar) to the end of the dotted entry line.

We can now complete our sales invoice form:

↑ ABC Company Sales Invoice ↑

Customer name ..... Date ..../..

Street address .....

City .....

Quan ..... # ITEM ..... Price ..... \$ Total ..... \$

Tax ..... \$

Invoice Total \$ ..... \$

Look at the four new PROMPT fields before you type them onto your screen. The PROMPTS 'Price', 'Total', and 'Tax' are simple MONEY fields. 'Price' calls for an entry of 7 characters (6 dots and the \$ sign). 'Total' allows for a 7 character entry (6 dots and the \$ sign), and so does 'Tax'. The PROMPT for 'Invoice Total' may be confusing. Here the PROMPT ITSELF ends in the \$ sign. This is perfectly O.K. as long as you're careful.

For 'Invoice Total \$' the trailing \$ sign in the PROMPT simply allows the final form of the MONEY entry to read:

Invoice Total \$ 125.75 instead of  
Invoice Total 125.75

This is a purely cosmetic option for the program designer. As a trailing sign, of course, the \$ symbol could be ANY symbol appropriate to the currency you are using. Only the \$ sign at the END of the dotted line MUST be the \$ sign as this is what tells CODEWRITER it's handling MONEY.

As you can imagine, you need to be especially wary of accidental dots in your PROMPTS where MONEY is involved.

## CODEWRITER CONCEPT

### PROMPT FIELD TYPES

**ALPHANUMERIC** (letters, numbers, symbols)—need NO special sign at the end of the dotted line. Ex. Name ..... They CANNOT be a part of a calculation.

**DATE**—may use ANY PROMPT but MUST use the input form .././.. as in Member since .././.. They CANNOT be part of a calculation.

**A GENERAL CAUTION**—Do not use the comma (,) colon (:) or semicolon (;) as part of your screen designs. These symbols confuse the file handling operations of both your CODEWRITER system and your CBM 64. To prevent problems, these keys are DISABLED when CODEWRITER is in use.

**NUMERIC** (numbers only)—may use ANY PROMPT but MUST use the # sign at the end of the dotted line. Ex. Amount .....# They CAN be part of a calculation.

**MONEY** (numbers only)—may use ANY PROMPT but MUST use the \$ sign at the end of the dotted line. Ex. Price .....\$ They CAN be part of a calculation.

**BOTH A NUMBER AND A MONEY** field need at least two characters to define their length. For example, the fields CASH PAID .\$ or NUMBER USED .# both have two characters (the dot and the sign) following the prompt. Use at LEAST two.



## SOME REAL CODEWRITER POWER

Our sales file can be much more than an electronic invoice system. Let's get down to some real PROGRAM DESIGN. By adding six additional fields to our screen, the CODEWRITER sales program can become a very efficient CREDIT JOURNAL while giving up to date reports on both TOTAL ACCOUNTS RECEIVABLE and TOTAL SALES. (Not bad for a first effort!)

Here is our screen with the six new fields:

```

      ↑ ABC Company Sales Invoice ↑
Customer name ..... Acct # ..... Date ..../../.
Street address .....
City .....
                                INVOICE # .....
Quan .....# Item ..... Price .....$ Total .....$
                                Tax .....$
                                Invoice Total .....$
PAID ON ACCOUNT .....$ INVOICE BALANCE .....$
↑ .....↑
TOTAL ACCOUNTS RECEIVABLE .....$ TOTAL SALES .....$

```

The six new fields each have a specific job. Here's a look at them one by one:

Acct # .....

This allows each ABC Company customer to have his own identity—even if names are alike. We have allowed for 5 places. Notice there is no # sign after the dotted line. There are two reasons for this; First the # sign would limit us to NUMBERS ONLY. Some account numbers use both letters and numbers (as T1450 etc.) to give greater variety using the fewest places. Secondly, using the # sign requires a bit more computer memory. Whenever CODEWRITER sees this sign (or the \$), it holds extra computer memory space aside in case the information in the field would be needed for use in a CALCULATION. Since we aren't likely to use account numbers in any calculation, why not save computer memory?

**Invoice # . . . . .**

This five-place field identifies a PARTICULAR SALE to our ABC Company Customer. By using BOTH the Acct # AND this Invoice #, we allow our CODEWRITER program to group together, in its memory, ALL the sales to the SAME Account number. We'll show later why this helps. Again, we left off the # sign (for the same reasons as the Acct # example above).

**PAID ON ACCOUNT**

This seven-place \$ field will be used to record customer payments against the particular invoice which is on the screen. We used the \$ sign because money is involved AND because this field WILL be used in a calculation. We'll explain the calculation function later.

**INVOICE BALANCE**

This field will hold the DIFFERENCE between the amount shown on screen as 'Invoice Total' and 'PAID ON ACCOUNT'. Again, the \$ sign is used because this field will always involve money. Also we'll use 'INVOICE BALANCE' as part of a calculation. Our CODEWRITER program will be designed to calculate this amount automatically.

**TOTAL ACCOUNTS RECEIVABLE**

This \$ field is intended to give a RUNNING GRAND TOTAL of all the balances carried in the field 'INVOICE BALANCE'. We have placed this field on the screen below the ===== header line to help show that the amount is a total of ALL the invoices in the file rather than the particular invoice on the screen.

**TOTAL SALES**

Again, this \$ field is a FILE WIDE GRAND TOTAL of ALL sales rather than relating to the invoice on the screen. We'll show later how to design CODEWRITER programs to perform the grand total function.

Our sales invoice is now complete. Of course a real sales invoice would have more lines to enter sales items and prices, but for our example this is enough. You are perfectly free to adjust the screen until your invoice form looks as close to our example as you wish to follow the manual.

Now the real magic of CODEWRITER will come clear. You may have been asking yourself "What does drawing a screen form have to do with writing a program?" The answer in the CODEWRITER system is "almost everything". CODEWRITER will "read" the screen we have just created and develop AUTOMATICALLY the entire file structure needed to make our program run. All the PROMPTS will be saved in the right places. The 'dates' will be saved as 'dates', 'money' as 'money', etc. Most of the program designer's work in creating this program is over!

### SCREEN READING

Once you're satisfied with the screen on your computer, press ESC to begin the "reading" we just spoke about. The screen will go blank for a moment and our sales invoice form will be replaced by the words "READING SCREEN". In a moment our screen will return.

Certain PROMPT fields on the screen will be HIGHLIGHTED in REVERSE and a question will appear at the bottom of the screen. CODEWRITER will skip over any LABELS, date and ALPHANUMERIC fields we've created and ask questions only about fields which contain NUMERIC and MONEY information.

The program designer is asked here whether a particular PROMPT field is to be "keyboard entered" or "program calculated". This simply means: "Do you wish to have the program operator enter the information the PROMPT requests or do you wish to have CODEWRITER itself calculate the response?"

NOTE: The third choice, g for GLOBAL, allows your CODEWRITER program to accumulate TOTALS from ALL the records in the file. More about this later.

In our sales invoice example answer the following as the fields are HIGHLIGHTED in REVERSE:

- |       |  |  |
|-------|--|--|
| Quan  | <input type="text" value="enter 'k'"/> | The operator must enter this from the (k)eyboard   |
| Price | <input type="text" value="enter 'k'"/> |  |
| Total | <input type="text" value="enter 'p'"/> | The CODEWRITER (p)rogram can calculate this amount by multiplying "Quan" times "Price". Why make extra work for the operator.                                  |
| Tax   | <input type="text" value="enter 'p'"/> | As long as the sales tax rate is constant for all items, your CODEWRITER created program will recall the rate as a percentage and multiply this by the "Total" |

Invoice Total \$  CODEWRITER will write program lines to direct the adding of "Total" to "Tax"

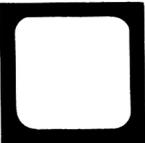
PAID ON ACCOUNT  The program operator will enter this amount.

INVOICE BALANCE  Your CODEWRITER (p)rogram will calculate this

TOTAL ACCOUNTS RECEIVABLE  for this (g)rand total.

TOTAL SALES  CODEWRITER will ACCUMULATE the Invoice Total amounts and show the TOTAL whenever the operator looks in the SALES FILE

Once all the appropriate fields have been designated either "k", "p", or "g" by the program designer, CODEWRITER will return to the Screen Format Generator menu where the following choices are offered:



(e)dit screen format  
(c)hange screen format  
(s)ave screen  
(l)oad screen  
(x) Exit to System Creation Menu

For now, do NOTHING. Here is what the menu options mean:

**EDIT SCREEN FORMAT**—If the program designer wished to make **ENTRY CHANGES** in the screen, he would use this option. By **ENTRY CHANGES** we mean changes in the **KIND** of information to be entered, such as adding or subtracting a **PROMPT**, or in the **SPACE** allowed to respond to a **PROMPT**.

Once the 'e' for edit is selected, the current screen in memory will re-appear. CODEWRITER will then allow ANY changes to be made to the screen as though it had just been typed in. All 'k' or 'p' choice information needs to be **RE-ENTERED** before leaving the Edit Screen option.

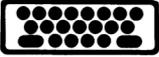
**CHANGE SCREEN FORMAT**—This option is strictly for **MOVING** existing screen information around. No new fields may be added or existing fields or labels removed. More about "Change Screen" later.

**SAVE SCREEN**—This option allows the **CURRENT** screen in memory (the one we just created) to be saved to the disk in the drive. More about "Save Screen" later.

**LOAD SCREEN**—This option allows a previously saved screen to be loaded from the disk in the drive. Thus **ALREADY CREATED** programs made with **CODEWRITER** could be modified later by loading just the screen with this option and then going back to the **Edit Screen Format** option to continue creating a **NEW** program. To simply **VERIFY** proper screen save, **Change Screen** can work better. More later.

**EXIT TO SYSTEM CREATION**—This option starts things over from the beginning. **BE CAREFUL HERE!** If you choose the exit option **BEFORE** saving your screen, the screen will be **LOST**.

Even though our current sales invoice screen shouldn't need any changes, let's choose the **CHANGE SCREEN** option anyway—just to watch how well it works.



type 'c' here

You should see an instruction screen to explain the workings of "Change Screen". This is for future reference. Read over the screen and then press **RETURN**.

Once again the sales invoice form should appear. The **LABEL** ↑ **ABC COMPANY SALES INVOICE** ↑ should have the cursor at the **FIRST POSITION**. Let's say you weren't satisfied with the way the **LABEL** was centered on the screen. Press the **RETURN** key and the **LABEL** should change to **REVERSE** screen image.



A field **SHOWN REVERSE** this way is ready to be **MOVED**. Simply use the cursor keys and move the **LABEL** anywhere on the screen you wish! Should your moving label bump into another field on its journey around the screen, **CODEWRITER** will automatically **JUMP** the label to the next empty area in the direction it was being moved. Once you're satisfied with the position of the moving field, simply stop and press the **RETURN** key. The field will revert to the normal print mode from **REVERSE**. **ALL** screen fields can be moved in the same way.

Press **ANY** key (except **RETURN**) and you'll skip to the next field where the process can be repeated as often as you like. With each pressing of a key the cursor will move to the beginning of the next field. The cursor will move over the fields in the **SAME ORDER** in which the fields were **FIRST ENTERED**. Check your screen instructions for the correct method to **BACK UP** through preceding fields.

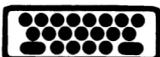
Making "Changes" can lead to some confusion. Remember the Change Screen routine does NOT alter any of the logic of the screen CODEWRITER has already read. Thus, if you move the fields all over the screen, your CODEWRITER program will continue to prompt for the operator information in the SAME ORDER in which you FIRST typed the fields in. If you'd like the NEW screen positions to dictate the NEW order of operator entry of data, you'll need to "read the screen" AGAIN with the Edit Screen option.

To make permanent changes with Change Screen, one should:

1. Move the fields around any way you wish from Change Screen.
2. Once changes are complete, press ESC to return to Screen Format Generator.
3. Choose Edit Screen and your NEWLY ALTERED screen will appear.
4. Make any ENTRY CHANGES (see Edit Screen) you wish to further alter the screen if needed.
5. Step through the 'k', 'p', or 'g' choices again. Once complete, you'll be back to the Screen Format Generator menu.
6. Choose "Save Screen" to save your new form permanently to the CODEWRITER disk work space. NOTE: If you have already saved a screen in an OLD order and now wish to save the screen with NEW field positions, give the NEW screen a NEW file name.

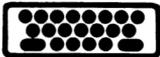


As we don't require any permanent changes to our example program, press fl to leave the Change Screen option. Here CODEWRITER warns us to be sure to save the screen. Once back at Screen Format Generator, we are ready to save our sales invoice screen.



Press 's' here

CODEWRITER will ask the program designer to give a NAME to the screen. A maximum of 10 characters is allowed and, as usual, simple, appropriate names are best. In this case, the name of the screen becomes the name of the PROGRAM to be created by CODEWRITER. Do NOT use a slash (/) or a dot (.) as part of a screen name. Also, a screen file name must be lower case.

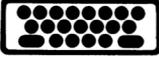


enter 'invoice' and press RETURN

It would be nice to VERIFY that our screen has been saved correctly. Since we are now back to the familiar Screen Format generator menu, we can VERIFY quite simply.

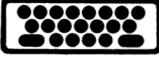
press 'l' for load and RETURN

The "load" option will ask for the 'screen file name'. We chose 'invoice' so:



type 'invoice' and press RETURN

The disk in the drive should spin and stop. Next the screen format Generator menu appears. We could choose Edit screen to see our newly loaded screen, but this would force the 'k' and 'p' choices again. Instead we choose Change Screen:

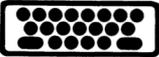


press 'c' and RETURN



From Change Screen we are shown our sales invoice form again which proves it has been saved correctly. To exit Change Screen we press the fl key.

Screen creation is complete and we may now continue with CODEWRITER program design.



press 'x' and RETURN

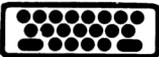
We get one last warning to save our screen. Quite a worrier, that CODEWRITER!

### CREATE DATA ENTRY SYSTEM

From the current menu we have the choices:



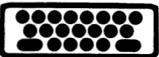
```
s  create screen layout
a  create application
x  return to Main Menu
```



press 'a' and RETURN

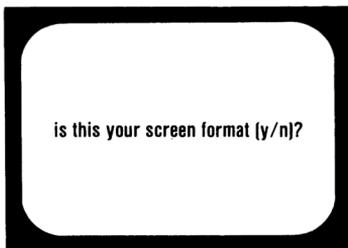
CODEWRITER now announces that it will "produce the basic code for a program you design". You may now enter a name (maximum 25 characters) and press RETURN. (The name will follow the credit: PROGRAM Design by

You are next asked for the name of the screen file. Be EXACT here so the system can find our much maneuvered screen:

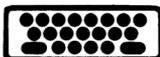


enter 'invoice' and press RETURN

After a bit of whirring from the drive, the sales invoice screen should re-appear with a few changes; The arrows around the LABEL ABC COMPANY SALES INVOICE should be gone. Also, any fields we designated (p)rogram calculated should have only a single dot following the PROMPT. You'll be asked:



If the screen you see is correct



Press 'y' and RETURN

If you press 'n', you'll be returned to the request for "screen file name" for another try at finding the screen file.

#### **WHICH DRIVE FOR DATA**

On your system, the prompt "Which drive for data DOES NOT appear. You'll be hearing soon how to upgrade your system to use more than one drive—for considerably more file space and power. We have the reference in the manual for the future. You may continue here if you like or SKIP to page 22.

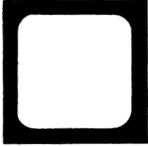
The choice is important. Remember, you are now creating a data entry program to control information. The information itself (the data) need NOT be on the same disk as the program which controls it. Keeping the control program on one disk and the data on another MAXIMIZES the amount of data you can control. On the other hand, where disk capacity is enough and the convenience of both program and data on a single disk is important, a one-disk system works fine.

Remember, the question means "which drive for data" when your PROGRAM IS COMPLETE AND RUNNING. (Users with two-drive CODEWRITER systems will NOW have their program disk in the second drive, but it will RUN in the first drive when it's finished. The "data" can be on either the first or second drive.)

For our example program, enter the appropriate number for the FIRST DRIVE.

## THE GRAND TOTAL FIELDS

The next CODEWRITER request will be to define what kind of GRAND TOTALS we want in the program being designed. In our example, the prompt screen will say:



Please specify which field is to be accumulated in this Grand Total field.  
(k)eyboard entered, which would be calculated by the (p)rogram,  
and which would be a (g)rand total of some other field.

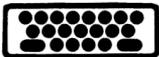
Total Accounts Receivable:

What does all that mean? When we first designed the screen for our sales invoice, we included a total of 14 different FIELDS. We then specified which of the information inputs would be (k)eyboard entered, which would be calculated by the (p)rogram, and which would be a (g)rand total of some other field.

The CODEWRITER system is now ready to learn how the program designer wishes to CALCULATE the information on the screen. To make things easier, CODEWRITER has ABBREVIATED the names of the screen fields. Thus the FIRST field on the screen to be designated (k)eyboard (e)ntered becomes ke1, the second becomes ke2, and so on. Naturally enough, the first field we chose to designate (p)rogram (c)alculated becomes pc1 to CODEWRITER.

Now, back to Grand Totals. CODEWRITER is asking which screen field is to be accumulated and displayed as a Grand Total after the prompt "Total Accounts Receivable". Inside the parentheses are the choices: kel to ke10, pc1 to pc4, or 'list'.

Since it's difficult to remember WHICH field we designated as the FIRST program calculated (pc1), etc. CODEWRITER offers the 'list' option to display all our choices.



Type 'list' and RETURN

You should now see the following on the screen:

**Keyboard Entered Fields:**

ke1= Customer Name	ke2= Acct #
ke3= Date	ke4= Street Address
ke5= City	ke6= INVOICE #
ke7= Quan	ke8= Item
ke9= Price	ke10=PAID ON INVOICE

**program calculated fields:**

pc1= Total	pc2= Tax
pc3= Invoice Total	pc4= INVOICE BALANCE

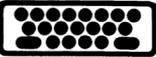
**grand total fields:**

gt1= TOTAL ACCOUNTS RECEIVABLE	gt2= TOTAL SALES
--------------------------------	------------------

Again, back to the CODEWRITER prompt we're trying to answer. We want our program to make it easier to get useful information. Which of the screen prompts we designed will ADD UP TO a GRAND TOTAL we can call "TOTAL ACCOUNTS RECEIVABLE"? Study the list. "Invoice Total"? Maybe, but what if we receive a payment from a customer? The "Invoice Total" would, of course, remain the same after a payment, but the amount the company is owed (its receivables) would go down.

The correct answer is INVOICE BALANCE. Obviously, if we had a Grand Total of the INVOICE BALANCE amounts from ALL invoices we could call this figure our TOTAL ACCOUNTS RECEIVABLE.

The 'list' should still be on your screen. We can see that INVOICE BALANCE is abbreviated by CODEWRITER to pc4.



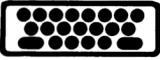
**press RETURN**

Again we see prompt:



Please specify which field is to be accumulated in this grand total field.  
(ke1—ke10, pc1—pc4, or 'list')

**TOTAL ACCOUNTS RECEIVABLE:**



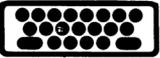
type 'pc4' and press RETURN

This tells our CODEWRITER program to accumulate ALL the INVOICE BALANCE amounts from the entire file of invoices and show the total in TOTAL ACCOUNTS RECEIVABLE on the screen. Whenever the operator of our program looks at ANY invoice in the ABC Sales file, he or she will always see this grand total on display.

The next CODEWRITER prompt asks for the field to accumulate as;

### TOTAL SALES

This should now be easy. Type 'list' again. This time, of course, 'Invoice Total' is correct as the amount to be accumulated as TOTAL SALES. Press RETURN to go back to the prompt.



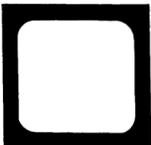
type 'pc3' and press RETURN

### COMPUTED FIELDS

The CODEWRITER screen now requests the computations for 4 computed fields. You'll be given an entire second screen of information as to what this means and an entire screen as to what is meant by 'self referencing' fields.

As before, these screens are reminders for later. We'll explain the procedures here in detail. Read the two screens and press RETURN.

The screen now shows:



```
Computed field #1
calculation for 'Total'
Type 'list' for field numbers

pc1=
```

This is where you learn to be a Program Designer. Designing the screen was the most creative aspect of the job. Now comes the real power.

Type 'list' to see your choices. As you look at the list of prompt fields and their CODEWRITER abbreviations, think. What is the DEFINITION of "Total"? In our invoice design, "Total" (pc1) means 'Quan' (ke7) multiplied by 'Price' (ke9).

We "design" this definition with CODEWRITER by saying:



$pc1=ke7*ke9$

===== **CODEWRITER CONCEPT** =====

As with most computers, the four basic arithmetic functions are:

- + means add
- means subtract
- \* means multiply
- / means divide

CODEWRITER also allows the use of ( ) to isolate formula components.

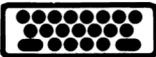
Parentheses are used to ISOLATE the calculations inside them for SEPARATE COMPUTATION within a formula. An easy example would be:  $pc1*pc2+(ke3-ke6)$  which means—First multiply pc1 by pc2 and then add to this result the difference between ke3 and ke6.

While CODEWRITER will detect SOME mathematical errors (such as forgetting a closed parentheses after using an open parentheses), it CANNOT prevent all instances of incorrect math from getting into a program. You'll be offered a chance to VERIFY a formula after you type it in. Once verified, however, CODEWRITER will try to audit what it can and then ACCEPT what you wrote. Please be careful.

Here are the remaining program calculations for our Sales Invoice design and an explanation of each. Follow the screen commands to enter these:

-----  
program calculation

-----  
meaning



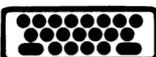
$pc2=pc1*.06$

Tax (pc2) is 6% of the Total (pc1) to the invoice. Thus, we multiply pc1 by .06 to find Tax.



$pc3=pc1+pc2$

Invoice Total (pc3) is simply Total (pc1) PLUS Tax (pc2)



$pc4=pc3-ke10$

Invoice Balance (pc4) is the result of Invoice Total (pc3) MINUS PAID ON INVOICE (ke10).

As usual there are a few rules to keep in mind. We'll try to be concise:

1. Calculation definitions must deal in KNOWN IDEAS. Thus, you cannot enter  $pc2=pc6-ke3$ . Can you see why? Calculations are defined in the SAME ORDER in which they appear on the screen (top to bottom, left to right). Thus, if you are defining  $pc2$  you CANNOT have defined already  $pc6$ —making  $pc6$  an UNKNOWN IDEA. This quandry is easier to avoid than you may think. Simply design your screen so that your input prompts PROGRESS in logical order (price before total, payment before balance, etc.). CODEWRITER will handle things from there.
2. Program calculations are the HEART of a good design. Use them well. They may contain ANY combination of  $pc$  fields,  $ke$  fields and even  $gt$  fields (subject to rule 1). They should be limited to 25 characters in overall length.

### SELF REFERENCING FIELDS

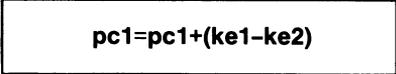
There is a bit more power in CODEWRITER calculations. The Self Referencing field may seem abstract and confusing at first, but it's JUST PERFECT for some jobs. Where the program designer wishes to HOLD a PREVIOUS value while calculating a new one, he needs a Self Referencing field.

An example is in order. In an inventory program, a field named BALANCE ON HAND will usually be designed to depend on two others like QUANTITY IN and QUANTITY OUT.

Lets assume that QUANTITY IN is  $ke1$  and QUANTITY OUT is  $ke2$ , while BALANCE ON HAND is  $pc1$ .

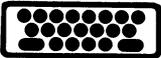
If we used a formula like  $pc1=ke1-ke2$  (which might seem logical), our inventory would be a disaster. Can you see why? The field of BALANCE ON HAND would always contain ONLY the LATEST results of the CURRENT difference between QUANTITY IN and QUANTITY OUT.

What's needed for a field like BALANCE ON HAND is a way to REMEMBER the current value, hold it, and then COMBINE it with a new value. Though many methods for doing an inventory exist, one approach might be:


$$pc1=pc1+(ke1-ke2)$$

CODEWRITER sees this as Self Referencing since the  $pc1$  appears on BOTH sides of the = sign.

Another use for self referencing is in a pure "counting" field. Since all 'pc' fields are automatically calculated EACH TIME a record is looked up by the operator, a field named "Record Access Times" (as  $pc5$  for instance) could be designed to count the number of times a records was looked up by defining it as:



$pc5=pc5+1$

CODEWRITER will automatically create a special file for self referencing fields whenever it sees a calculation with the SAME pc on BOTH sides of the = sign. The program designer needn't do anything but write the formula.

Because the self referencing file will take extra space on the program disk, CODEWRITER will ask the designer to "confirm" that this unique field is what the designer truly wishes. Simply type 'c' to confirm as directed.

REMEMBER—The self-referencing field is for Program Calculated (pc) fields only. The CODEWRITER system contains special features for AUTOMATIC UPDATING of Keyboard Entered (ke) fields. These features are explained later, under "UPDATE DATA", in the instructions for using ANY CODEWRITER designed data entry program. Don't worry if "self-referencing" is not quite clear yet. Just keep in mind the following:

1. "self-referencing" means holding an existing value while combining it with a new one.
2. A self referencing field is ALWAYS program calculated
3. A keyboard entered field can do ALMOST the same thing another way.

One last thing. Once defined, a self referencing field MUST have some opening value (even zero) to function. This needn't be done by the designer, but must be done the FIRST time the program operator encounters the field on the screen. CODEWRITER anticipates this. Should a program operator pass a self referencing field the first time WITHOUT entering a value, the prompt "You must enter something" will appear at the bottom of the screen. Again, a zero entry is fine.

### **THE NUMBER OF RECORDS NEEDED**

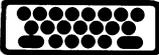
Once field calculations are completed, CODEWRITER will ask:

"What is the maximum number of records you want in the data file (50 to ----)?"

This calculation is made automatically by CODEWRITER and depends on the amount of information in a screen design.

CODEWRITER calculates the maximum for you and asks how many you're likely to need in your file. CODEWRITER will then reserve the correct amount of space on your program disk. Remember that specifying the maximum here will FILL the program disk. Where you would like MORE than one program on the same disk, ask for the FEWEST records practicable for your use.

For our Sales Invoice example, a small record file will do.



enter '50' and press RETURN

Next, we are directed to "Type in the program title" and are allowed 30 characters to do so. The program 'title' is NOT THE SAME as the 'Screen file name' we chose earlier. This 'title' is cosmetic only and will merely be printed above the menu CODEWRITER will automatically create for your programs. The 'title' should simply describe what your program DOES.

Enter something like



'ABC SALES RECORDS'

and press RETURN

### THE KEY FIELD

You should now see on the screen the following questions:

"Which field is the key field (type 1 to -- or 'list' to list fields)"

The "key field" is more computer jargon for a not too difficult idea. The program which CODEWRITER is creating from our design will store records in a file and then get them back as we need them. To find a particular record (screen), the program conducts an electronic 'search'. The program can simply look at every record in file until it finds what we need, or it can go MUCH MORE DIRECTLY to the record in question.

The difference is having a "key" field to search for. Where one field on our screen record is designated the "key", the CODEWRITER created program can go to a SPECIAL INDEX of "keys" it had previously set up. In a flash the needed screen appears.

There is no need for special computer knowledge to choose the "key" field. The "key" is simply the one piece of information (field) MOST LIKELY TO BE LOOKED UP when searching a file.

As an example, in a sales invoice file it is very likely that records will be searched by 'Customer name' most often. Perhaps, in another case, the screen form for the invoice contained a 'customer number' or 'account number'. Certainly either of these would make a good "key" field as well.

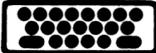
For the moment, type 'Li' and press RETURN

You should see a screen like this:



Keyboard entered fields:

- |                  |                     |
|------------------|---------------------|
| 1. Customer name | 2. Acct #           |
| 3. Date          | 4. Street Address   |
| 5. City          | 6. Invoice          |
| 7. Quan          | 8. Item             |
| 9. Price         | 10. Paid on Invoice |



Our choice is limited to the 10 fields designated 'keyboard entered'. A 'program calculated' field can NEVER be a "key". The CODEWRITER system has numbered our fields from 1 to 10 and has kept track of the numbers. Thus we can choose the "key" by entering the number only. Let's make 'Invoice #' the key.

enter '6' and press RETURN

#### **MORE ABOUT KEY FIELDS**

You may search by ANY field on a record screen. The key field is simply the fastest and most direct way to search. To design the BEST POSSIBLE key field, keep one rule in mind; The best key in a record is the most unique key.

For instance, in our invoice example the 'Acct #' key may be REPEATED in many records (where the same customer buys many different times, for instance). Since the 'Acct #' entered is the SAME for many records, each time a 'search' on the key field is done many records will 'qualify' in the search. This will work, but is not the MOST EFFICIENT way.

Try to devise a key which will be unique to a SINGLE record. In our example, the Invoice# is best. This number will be DIFFERENT for each record entered.

Again, any keyboard entered field may be the 'key' and a key which can refer to multiple records is O.K., but unique is best.

Since many CODEWRITER applications will involve money, we can use a bit more advice on the subject. Here are a few tips:

1. CODEWRITER will allow an operator to enter simply 23. and this will print as 23.00

2. An amount with NO numbers to the left of the decimal place as in .10 will be printed later as 0.10
3. Where the program designer wishes to make sure that money amounts line up top to bottom with the decimal points EVEN, care should be taken to see that the DOTTED LINES for money justify TO THE LEFT. For example:

```
left
.....$      (7 places)
.....$      (9 places)
...$         (4 places)
```

will result in a column of money amounts with the decimals in line TOP TO BOTTOM even though the \$ signs vary. The fact that the dotted entry lines are justified LEFT will accomplish this.

### **KEEPING OUT GARBAGE**

We are almost finished with program design. This last section is really optional, but it can be quite important.

Any collection of information can be made most valuable to the extent it can be kept PURE. That is a file on stamps should not contain an occasional recipe and a PROMPT field for price should not allow letters to be typed in, etc.

Without some attempt at keeping out 'garbage' entry, a file can become an awful mess and lose a lot of its value.

You should now be looking at the first of two screens which show how the CODEWRITER system allows the program designer TRAP OUT ERRORS in operator entry.

Like the other instruction screens on the CODEWRITER disk, these are for future reference. Let's go through them now for more detailed understanding.

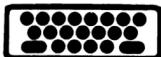
Once past the two screens, CODEWRITER will bring our sales invoice screen back into view and begin to HIGHLIGHT each of the KEYBOARD ENTERED fields. At the bottom of the screen there is a prompt line saying: Reject if: at the same time as ONE FIELD is HIGHLIGHTED above.

The program designer is being asked, "What will not be accepted?"

CODEWRITER offers a complete arsenal of weapons to keep out nonsense and a very good system for letting a program operator know when something is wrong.

In order to best use your Reject if: weapons, we'll go through the entry process together. Remember, you can always type 'help' to see all the types of data traps again on the 2 screens. You SHOULD study the screens as we go.

You'll see HIGHLIGHTED on the screen 'Customer name . . . . . etc.' and 'Reject if:' below.



enter 'no entry' and RETURN

This means that we have DEMANDED SOME ENTRY by the operator of our program. Since 'Customer name' is quite important, the operator musn't leave it blank or the sales record could be confusing.

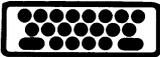
Once 'no entry' is typed and RETURN pressed, you'll see:



Error Message?  
(cr= \*\*\* you must enter something \*\*\*)

CODEWRITER is asking the program designer to write a message to the program operator EXPLAINING that the mistake 'no entry' was made. The "cr= \*\*\* you must enter something \*\*\*\*" means that if the program designer wishes, the message "you must enter something" will be entered AUTOMATICALLY by CODEWRITER as a response to the 'no entry' error. (the cr means (c)arriage (r)eturn or just RETURN)

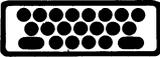
Let's write our own error message:



enter 'You must enter customer's name.' and RETURN

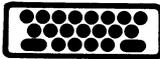
As you'll see the SAME field will remain HIGHLIGHTED and the Reject if: message will appear again. Why? Because MORE than one error could be made in the same field entry.

Let's say we want to prevent an entry which is TOO LONG. The 'name' field is 30 spaces. We can use the edit feature of CODEWRITER to automatically reject an entry longer than 29 (in this case). The rule is; Where you wish to restrict length, allow the space involved to be at least ONE SPACE MORE than the entry you wish to reject.



enter 'length>29' and RETURN

Your CODEWRITER program will then reject all entries MORE THAN 29 spaces in length. (The symbol after 'length' above means 'greater than'.) This prevents an operator from typing more information than your screen form can accept. Regardless of any edits you provide, your CODEWRITER program will automatically sound a BELL when an operator tries to type PAST THE BOUNDS of your screen format size for a given field (i.e. more than 10 spaces in a 10 space field).



enter 'customer name cannot be over 29 spaces' and RETURN

### CREATIVITY IN PROGRAM DESIGN

The choosing of edits and operator messages to trap out errors is where the personality of the program designer really comes through. The "attitude" of the created program toward its user, and the general need for accuracy, is built at this point.

Rather than go through all of the fields in our example program, we'll offer, instead, some suggested "edits" and messages. Once you feel comfortable with the process of edit control, by all means devise your own.

Field Name	Reject if: [syntax]	Meaning	Message
Acct #	contains 'ab'	Invoice #CANNOT contain 'ab'	"ab accounts only in file 5."
Acct #	length<5	Acct #'s MUST have 5 digits	"The Acct # entered is too short"
INVOICE #	no entry	As KEY FIELD, it MUST be entered	"Please include the invoice #."
Quan	not numeric	A quantity MUST be entered as a number	"Please express quantity as a number"
Price	> 10000	No number OVER 10,000 will be accepted	"Items costing over \$10,000 use form 3"

(Note: Though the following aren't in our example program, they help to illustrate the edit process.)

Last Name	> 'D'	No name beginning with D or later will be accepted	"This form for A to C names only"
Part #	=300	Don't accept 300	"Item 300 has been dropped-see note 10"
SEX	<> 'male'	MUST be male	"Use male only for this survey"

WARNING: While edits can be COMBINED to test the SAME field for different kinds of operator errors, some combinations are LEATHAL—as they allow no entry at all (or eliminate a range of entries by mistake). For instance, >"a" rules out EVERY lower case letter entry. (Can you see why?) And >100, when combined with <50, allows ONLY 50 to 100 to be entered.

By studying these examples as well as the two edit screens, you should be getting a good idea of the editing process. Remember, CODEWRITER will process as many or as few information edits as you wish. Don't leave edits out entirely, though, as they can be the "soul" of a good information file.

**PLEASE NOTE: If the "reject if:" syntax is still not clear, see Appendix A at the end of the data entry section in this manual.**

Once the edit section is complete, CODEWRITER asks if you would like a special "end of data entry" message to be used in your program. This message allows the program operator to either get a new blank screen form to fill in or return to the program menu.

The Program Designer is free to choose his own language here, but ONE bit of program LOGIC is automatic: If the operator presses the RETURN key at the end of filling in a screen, a NEW SCREEN will appear. And if "y" or 'yes' is entered, the program STOPS DATA ENTRY and returns to the menu. Examples of "legal" messages are:

"Are you ready to stop data entry y or n (RETURN = n)"

or

"To return to the Main Menu press "y", to continue press RETURN"

If you'd rather not bother to compose any special message, simply press the RETURN key and CODEWRITER will write its own message as shown on the screen.

This final CODEWRITER design choice is for date format.



Enter an 'a' for American or 'e' for European date format in your program.

Any field you designated 'date' (by entering .././.. to the screen) will automatically be evaluated by CODEWRITER for legal date entries.

**THAT'S IT!**

Once the correct date format is selected, your system is ready to create a separate program disk to contain your new application. The procedures to do this will vary depending on which computer you're using.

Check your USER NOTES CARD and be sure to follow the SCREEN MESSAGES that are offered by your CodeWriter system.

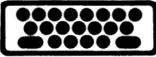
To run your program immediately simply type: run and press RETURN. Later, you run the program like any other piece of software—there is no further need for CODEWRITER until your next program design. Simply place your disk in the drive (after proper power up). The 'program name' is the name of your SCREEN FILE plus a /t . For example, if your screen file is named joe your program name will be joe/t . Both the screen file name and the t MUST be lower case.

The correct sequence for running your new software (once you have turned your computer off) is the following (We'll assume the screen file is named joe ):

```
load"joe/t",8  
run
```



NOTE: Where you have already created a program with CodeWriter you can load it into your computer's memory easily. First, find what's on the disk (load "\$", 8 then 'list'). You'll see a series of files with the same screen file name (but different suffixes). The '/T' file is the one to load. For example, our 'invoice' program would be loaded:



type load "invoice/t",8

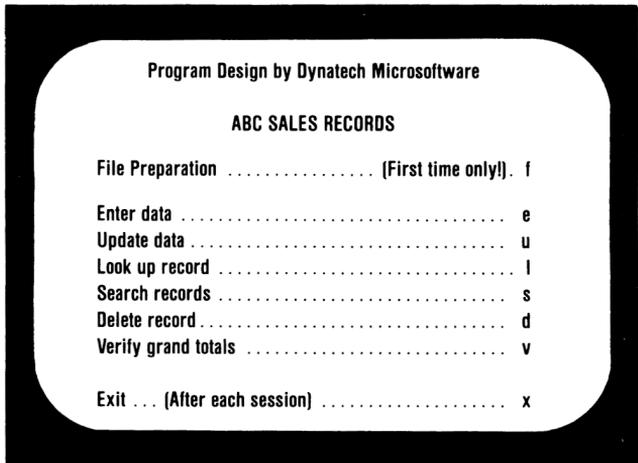
(Naturally, if you used a different "screen file name", substitute that name in front of the '/t' in the directory file name.)

Once the flashing cursor returns to the screen:



type 'run' and press RETURN

After a bit of disk activity, the Main Menu of your first CODEWRITER program should appear. Except for your name being used instead of ours, it should look like this:



Let's go through the menu options one at a time.

**File Preparation**—This is the CODEWRITER utility which prepares the disk designated to hold the data for the program. The File Preparation utility will create enough disk space on the data disk to hold the file the designer had requested. REMEMBER! This utility is used ONLY the FIRST TIME a program is run. Once there is data on a disk, the File Preparation utility will ERASE it to 'Prepare' a new file. Beware!

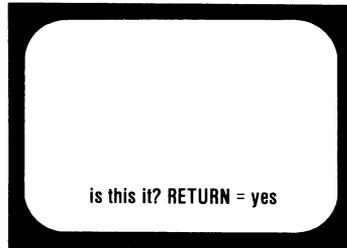


**Enter data**—This gives the program operator a new and empty screen form to fill in. At this point **ONLY** the **KEYBOARD ENTERED** fields are displayed (not program calculated, grand total or labels). To stop the 'Enter data' sequence mid-screen, press the **f1** key. Once a screen is complete, the operator will see a line showing how many records have been entered into the file and how many are left. Next the operator is asked whether the data entry session is complete. If not a new screen is shown. If so, the program returns to the **MENU**.

**Update data**—This program routine gives the operator a chance to change any information already entered into a screen record. The operator is asked to give the "key" information—that is the data entered in the field designated "key" by the program designer. Here's an example:

In our **ABC SALES** program, the "key" field is 'Acct #'. Thus, on Update the operator first sees a prompt asking for the 'Acct #' of the record to be 'Updated'. Once the Acct # of the record (invoice) is entered and **RETURN** is pressed, the program searches the disk for the record and displays it on the screen.

At the bottom of the screen, the prompt line displays:



If the record displayed is correct, press **RETURN**. You'll notice that now **ALL FIELDS** and **LABELS** are displayed. The results of program calculations appear and grand totals are listed where they were designed. (**NOTE**: If the record displayed is not correct, type 'n' and the program will continue to search.)

A new prompt now appears at the bottom of the screen. Using our Invoice Program as an example, the prompt reads:

Which field to update (1-10, 'list', **f1** to cancel, **RETURN** to save)

The prompt choices, inside the parentheses separated by commas, mean the following:

1-10—This is a choice of field numbers to **UPDATE** from field #1 to field #10. All are **KEYBOARD ENTERED** fields (the **ONLY** ones intended by the program designer for the program operator to be involved with).

list—Naturally, this gives the operator a list of the **KEYBOARD ENTERED** fields showing which **FIELD LABELS** belong with each of the 10 numbers. Once the operator sees which field # needs to be **UPDATED**, **RETURN** is pressed, the record screen returns, and the update is ready for a choice.





F1—At any time during UPDATE, the operator may press f1 and cancel the update process. This returns the main menu.

RETURN—To COMPLETE and SAVE the update to disk file, press RETURN.

This sequence illustrates the update process on our example program:

1. The operator notices that an incorrect price was used in a customer invoice already on file.
2. The UPDATE routine is called with "u" and RETURN.
3. The Acct #, 1005, is entered as called for.
4. The first record displayed is the right Acct # but the WRONG invoice, so 'n' and RETURN get a second invoice—which is correct.
5. 'List' is called to get the field # for 'Price', which is 9. The 9 is entered and the cursor appears at the 'Price' field—now erased and waiting for a new entry.
6. As soon as the new price is entered and RETURN is pressed, the screen action begins! Not only is 'Price' updated, but ALL the program calculated fields and grand total fields which in some way depend on the price amount are also updated and can be saved by pressing RETURN again once the revised screen appears.

Before we leave the UPDATE routine, there is one more valuable feature called "(m)ore and (l)ess". Here's an example from our ABC SALES program:

1. A customer wishes to make a payment on one of his open invoices. The operator goes through the update routine and finds that the field (#10) PAID ON INVOICE already contains a payment amount. The customer is making a second payment on the same open invoice.
2. Since in this case we don't want the amount NOW in PAID ON INVOICE to be ERASED and replaced with the current payment, the normal update won't do. (Let's say the amount currently in PAID ON INVOICE is \$15.00)
3. The operator chooses field #10 which places the cursor at PAID ON INVOICE and OVERWRITES the \$15.00 amount. Since the new payment is \$10.00 and we wish to ADD this amount to PAID ON INVOICE, the payment is entered as 10.00m for more. The 'm' ADDS the 10.00 to the previous 15.00 so when RETURN is pressed, the new PAID ON INVOICE amount reads 25.00 and once again all fields which relate to this change automatically.

Obviously, entering 'l' (for less) as in 35.00l would SUBTRACT 35.00 from the amount already entered.

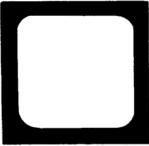
One last point; The m and l feature at first seems the same as a self referencing program calculated field, which holds an old value while calculating a new one. The two are different. The m and l feature works ONLY on KEYBOARD ENTERED fields while self referencing is ONLY for PROGRAM CALCULATED fields.

Keep "(m)ore and (l)ess" in mind for your future CODEWRITER applications. The feature is invaluable for inventory type programs especially.

**Look up record**—When the 'l' is used from the main menu, the user first sees the key field alone on his screen. The ENTIRE key field entry should be typed and then RETURN is pressed. Once a full screen appears, the user is offered a choice; If the screen record is correct, simply view as long as needed and then type x and RETURN. This will return to the main menu. If the first screen seen is NOT correct (there may be several with the same key), press RETURN and the program will search for another screen record with the same key.

**Search records**—This feature has two main purposes. One is to find screen records where the "key" field information is unknown. The second is to give the program operator a chance to view an entire SERIES of screen records which are LINKED by search boundaries the operator has chosen. Here's another example from our ABC SALES program:

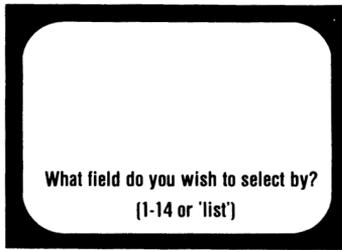
1. The operator wishes to find the invoice to "Abbott Jewelers" but does not know the Acct # to find "Abbott Jewelers" with the Look up command. The Search command is chosen instead.
2. After 's' and RETURN are pressed, the operator sees the prompt:



Scan all or selected records?(a/s)

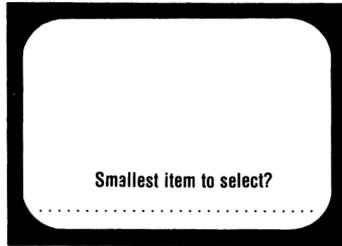
Since the operator doesn't want to see "all" of the invoices to find "Abbott", the 's' is pressed for 'selected' records.

3. Next the operator sees the prompt:



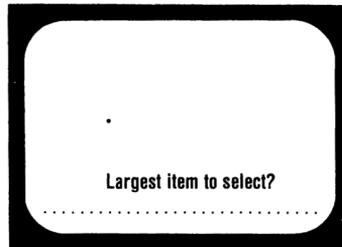
Here the operator types 'list' and sees a list of all 14 fields (except grand totals). The operator wants to search alphabetically so field #1, Customer Name, is selected for the search.

4. The next prompt reads:



Here 'Smallest' means lowest in the alphabet. Notice that the prompt offers 30 dots to fill in? That's because your CODEWRITER program "remembered" that field #1 was designed to have 30 characters maximum. The operator types 'Ab' which means that records with 'Customer Name' beginning lower in the alphabet range than "Abbott" (say Aaron, for instance) would be omitted from the search. Remember a lower case letter is 'lower' than its UPPERCASE counterpart (i.e. 'a' is lower than 'A').

5. The next prompt:



is answered; 'Abbott' so that nothing above 'Abbott' will be searched. The records within the range will be displayed one at a time along with the prompt:



To exit type x then RETURN.  
to continue RETURN

and so the operator simply presses RETURN until the desired record is displayed and then types x to halt the search.

A few more items concerning 'Search':

1. On an alphabetic search of, say, A to D remember that a lower limit of A is fine but D alone as the upper limit will leave out everything beyond D by itself. To search a file A through D, enter Aa or A and DZ as the two limits.
2. Remember that to a computer A is different from a. If you used capital letters in your fields, use capitals in your search limits.
3. Where a search field is a date, you'll be offered .././../. instead of dotted entries. You may search through a range of dates.

**Delete record**—This menu option removes records from the file disk. It works by asking the operator for the entry to the "key" field and then displays the screen record in question. by answering the "is this it" prompt with RETURN, the record is deleted.

**Verify grand totals**—Because of occasional instances of computer "rounding off" certain sums, the 'Verify' option is included. Simply enter 'v' and RETURN. No other entry is needed. All of the 'grand total' fields on the screen will be checked for accurate mathematic sums. The Verify option appears ONLY when a file contains grand totals.

**Exit**—This is simple, but can be easy to forget. After EACH session of data entry is complete, exit the program with THIS 'x' RETURN option. Do NOT simply turn the computer off. The 'Exit' routine in your program performs a number of very important "computer housekeeping" tasks which keep the data file ready for reliable use.

## A FINAL WORD

We have tried in this manual to show the major features of CODEWRITER and how these features work interactively to allow the Program designer to control information. We've shown some things in detail and only hinted at others—all by design. CODEWRITER is a tool, to be discovered rather than explained.

This CODEWRITER product is the first of a series aimed at making PROGRAM DESIGN more powerful and capable a function. We hope never to lose sight of the fact that your growth is our growth.

## Appendix A — The “Reject if:” rules

We thought it would be helpful to have the two “reject if:” help screens for your CODEWRITER program reproduced here for easier reference.

### screen one

#### GENERAL TESTS

Test name	example	meaning to operator
'no entry'	-	some entry required
'not numeric'	-	use only numbers here
'numeric'	-	don't use numbers here

#### DATA SIZE TESTS

Test name	example	meaning to operator
'length >'	length > 4	no more than 4 keystrokes allowed
'length <'	length < 7	no less than 7 keystrokes allowed
'length='	length=2	must NOT be 2 keystrokes
'length <>'	length < > 3	must be 3 keystrokes

### screen two

#### NUMBER TESTS

Test name	example	meaning to operator
'>'	> 100	must NOT be greater than 100
'<'	< 20	must be 20 or higher
'='	=631	must NOT equal 631
'<>'	<>17	must equal 17

#### CHARACTER TESTS (note single quotes)

Test name	example	meaning to operator
'>'	> 'd'	must NOT be after “d” in the dictionary
'<'	< 'jo'	must NOT precede “jo” in the dictionary
'='	= 'bill'	must NOT be “bill”
'<>'	<> 'male'	must be “male”
'contains'	contains 'abc'	must NOT contain 3-letter group “abc”

The various symbols used in “reject if:” syntax may not be familiar. Here's a detailed explanation. We'll take the tests in order.

Very often the most confusing aspect of the "reject if:" design is the backward or opposite nature of the prompt: the designer is asked to state what he does NOT want rather than what he does. Help is on the way.

**NO ENTRY**—Since the purpose of "reject if:" tests is to let the program operator know what is NOT accepted entry, "no entry" as a test is vital. When the program designer answers a "reject if:" with "no entry" the meaning is: "Don't skip over this field—it will be rejected if there is no entry."

Use the 'no entry' test when the field in question is the KEY FIELD. Without the 'no entry' test, the operator could leave the key field blank. With nothing in the key field, the ENTIRE SCREEN RECORD would be lost to the CODEWRITER system.

Anytime you wish to DEMAND SOME ENTRY to a field, use this test.

**NOT NUMERIC**—Again we must think in opposites. Where a field is designed for number type information only (i.e. quantity, number of days, part number, etc.) the designer should "reject" a "not numeric" entry by the operator.

This is critically important where a number will be part of a calculation. Obviously, if an operator answers a quantity question with "two" instead of "2", the calculation function will not work.

**NUMERIC**—Where a designer wishes ONLY TEXT to be entered to a field, the syntax is; reject if:numeric - The CODEWRITER program will not accept keystrokes 0 through 9 under this test.

**LENGTH >** —The meaning here is "length greater than". The ">" sign the computer symbol for "greater than". Literally, what is to the LEFT or LARGE side of the ">" is greater or larger.

In the case of "length", the "reject if:" meaning is the number of keystrokes (both spaces and characters) allowed for entry. Thus, where NO MORE THAN a 5 digit number is acceptable entry, use "length > 5" meaning "length greater than 5" as the correct test.

**LENGTH <** —No surprise here. The "<" symbol means "less than". Thus, where a particular part number MUST HAVE at LEAST 6 digits, for example, the test "length < 6" will prevent an operator from entering a number whose length is too short.

**LENGTH=**—Again, this is fairly clear. The meaning is "length equals". The test will screen out a SINGLE PARTICULAR LENGTH as "reject if: length=3". This test is not used very often, but COMBINED with some other test, may be useful.

**LENGTH <>** —The "<>" symbol is computerese for "does not equal". Where a designer wishes ONLY A SPECIFIC LENGTH of entry and nothing more OR less, this test is used. An entry of 6 keystrokes and NOTHING ELSE would be tested with, "reject if: length <> 6"

## QUANTITY and TEXT

Where the four symbols, “ > ”, “ < ”, “=”, and “ < > ” are used WITHOUT “length” and WITH numbers, they evaluate the QUANTITY INVOLVED rather than the number of keystrokes.

> —This still means “greater than”. Where you wish to prevent an entry of ANY HIGHER QUANTITY than 100, for example, the test is “reject if: >100”.

< —As you'd expect, the “less than” symbol works to prevent ANY LOWER QUANTITY than the designer wishes from being entered. To reject any lower entry than 50, for instance, the test is “reject if: < 50”.

= —As before “equals” seeks out a SINGLE QUANTITY ONLY to reject. Where, as an example, the ONLY wrong amount is 200, the designer tests for this with “reject if: =200”.

< > —The symbol means “does not equal” as before. Used without “length”, the test is to SEEK OUT A SINGLE CORRECT QUANTITY. Where the designer wants, say, only part 400 as a field entry, the test is “reject if: < > 400”.

Be careful with “does not equal” as a test. Since it accepts ONLY ONE quantity as correct, it cannot be combined with other quantity tests.

## LETTER TESTS

When used with quotation marks and letters, the “ > ”, “ < ”, “=”, and “ < > ” test for POSITION IN THE ALPHABET OR DICTIONARY.

> ‘ p ’ —In the example > ‘p’, the meaning is “greater than p” or “past p” in the alphabet. Using a SINGLE LETTER as we did limits the test to the FIRST LETTER in an entry. Thus the test ‘reject if: > ‘p’ ’ would TRAP OUT all words beginning with r or any other FIRST letter LATER THAN p in the alphabet.

Where MORE than one letter is used, dictionary position determines the “greater than” or “later than” test. The test ‘reject if: > ‘mac’ ’ would eliminate ALL WORDS later in the dictionary than a word beginning with ‘mac’.

< ‘ e ’ —Here the meaning becomes “lower than” a FIRST letter or group of FIRST letters in the alphabet or dictionary. To trap out ALL “d” words or lower in the alphabet, the test is ‘reject if: < “e”’. Thus, only “e” words or higher could be entered.

= ‘ frog ’ —As before “equals” looks for ONE THING only. Where, for some reason, the designer does not wish “frog” as an answer, the correct test would be “reject if: = ‘frog’”. Several of these tests can be combined on a single field to trap out a LIST of words or letters not wanted.

< > ' —As before, the “does not equal” symbol is used to trap out ONE SINGLE ITEM. Therefore if “tractor” is the only response the designer wishes to allow, it is demanded with “reject if: < > 'tractor'”. Also as before the “does not equal” test CANNOT be combined with others on the same field. It is seeking a single acceptable response.

**CONTAINS** ' —The “contains” test is used ONLY with words and letters within the CODEWRITER system. If, for instance, a particular letter or group of letters is to be tested for, “contains” will do the job.

Let's assume that a part number entry in some inventory analysis is “B1200” and the designer wishes to allow NOTHING from the “C” series (C1200, etc.) to get into the data by mistake. The correct test for the 'Part Number' field would be 'reject if: contains“c”'. All numbers and other letters would be ignored, but any entry containing “C” would be refused.

The “contains” test can also trap a CONTINUOUS GROUP of letters ANYWHERE in a word or sentence. Thus the test 'reject if: contains 'me' ' would trap out 'me' as well as 'men' BUT ALSO “some” and “stammer” (because they contain the 2-letter group 'me' ). 'Contains' is a powerful test. Be careful.

Keep in mind that with all the letter and word tests, an UPPER CASE letter is not the same as its lower case counterpart. You may have to test for BOTH kinds of entry to really be sure you keep out what you want out.

We hope this appendix makes the “reject if.” idea more clear.

Remember, while your program designs will be made more powerful by using these tests, they are optional. Use them as you are comfortable with them.

# CODEWRITER INDEX

Alphanumeric Prompt Fields	10, 13, 16
American Data Format	10
Arrows—Up/Down/Left/Right	6
Cassette Port	3
CodeWriter	
Backup Disks	3
Concepts	9, 13, 25
Loading	4
Manual	3
Safekeeping	3
Support	3
Column/Row Line	6
Cursor (CRSR)	6, 18
Data Disk	21
Data Drive	21
Data Handling—One & Two Drive	21
Date Format	9, 10, 33, 34
Dates (Data Entry)	9
Delete Record	40
Directory of Your Program	34
Disk Formatting	4
Disk Name	4
Disk Space	35
Dollar Fields	29, 30
Dynatech Microsoftware	2
Edits	32-33
Error Messages	31
Error Messages—User Written	32
Error Tapes	30, 32
European Date Format	10
Field Length Limit	26
File—Definition	6-7
File Preparation	35, 36
File-Wide Grand Total	15
Formatting—Caution	4
Garbage	30
Grand Total Display	24
Grand Totals	14-17, 23, 26, 36, 40
HELP (Command Option)	31
Key Fields	28-29, 36, 38
Keyboard-Entered Fields	16, 17, 22, 26, 38
Labels	7-8, 21
Largest Item—Defined	39
Letter Tests	43
List (Command Option)	22, 24, 26, 39
Main Menu (CodeWriter)	4
Messages—Error/Operator	32
Module	3
Money Fields	11, 12, 14, 16, 29
Number Fields	10-11
Numeric Prompt Field	13

## CODEWRITER INDEX (con't.)

Operator Messages .....	32
Program Calculated Fields, or PC Field .....	15-16, 22-26, 29-30, 36-37
Program Design .....	2
Program Generation .....	2
Program Generation Time .....	34
Program Name .....	19,36
Program Title .....	28
Prompts .....	8-10,21,23
Prompt Fields .....	12-13,16
Record	
As Screen .....	27
Defined .....	6,7
Records	
Search/Delete .....	38-40
Maximum/Fewest Number of .....	27-28
Reject-If Statements .....	32-33
Reject-If Statements Rules .....	41,42,44
Screen	
As Record .....	7, 27
Creation .....	5
Design .....	5-6
Editing .....	17
Fields .....	14
File Name .....	19, 34
Format .....	17
Format Generator .....	17-20
Label .....	7
Layout .....	5
Load .....	17-18
Options .....	6
Save .....	17, 19
Space .....	5
Self-Referencing Fields .....	26-27, 38
Single File Programs .....	7
Smallest Item—Defined .....	39
Software Protection .....	3
Space - Screen .....	5
Space - Maximum in Prompts .....	8
Support-Manufacturer .....	3
Update Data Option .....	36,37
Your Generated Program Menu .....	35
Your New Program .....	34