CLARION TECHNICAL BULLETIN

Bulletin #121

Clarion Memos

Overview

This bulletin contains information about Clarion memos and how to work with them in your applications.

Clarion Memos

To begin, let's clarify the definition of a Clarion memo. A memo is a special field that is not declared within the record structure of a file. Therefore, it is not a field within the record. Technically speaking, a memo is actually another record linked to a Clarion file record. Memos are stored in a separate file with the same name as the data file with the extension .MEM rather than .DAT (for the data file). When a data file record is read into memory, its associated memo is also loaded into memory from the memo file.

Memos are called fields because they are referenced the same way a field within a record is referenced. Also, when you create a memo with Designer, you use the same method that you use to create a "regular" field.

Memos on the Screen

When Designer generates a Form procedure that uses a file containing a memo, a TEXT field is used in the screen structure to input data into the memo. This is not to say that this is the only way to get screen input into a memo; but by using a TEXT field for a memo, you get basic word processing-type control for entering data into the memo. The TEXT statement in the screen structure to reference the memo directly could look like Figure 1.

```
ROW(7,4)     TEXT(5,60),USE(TST:MEMO1),HUE(15,4),SEL(0,7),LFT
```

Figure 1.

The use variable TST:MEMO1 is a direct reference of the memo for the data file with a prefix of TST. The parameters of the TEXT statement (5 and 60) indicate that the memo on the screen will be 5 rows by 60 columns. The LFT attribute at the end of the TEXT statement indicates word wrapping.

If you want to enter data into the memo without using a TEXT field, you can use a GROUP structure with the OVER(TST:MEMO1) attribute and create a variable for each row of the memo, as shown in Figure 2.

```
TSTMEMO1          FILE,PRE(TST),CREATE,RECLAIM
BY_ACCOUNT           KEY(TST:ACCOUNT),NOCASE,OPT
BY_NAME              KEY(TST:NAME),DUP,NOCASE,OPT
MEMO1                MEMO(300)
RECORD               RECORD
ACCOUNT              SHORT
NAME                 STRING(30)
                  . .

                  GROUP,OVER(TST:MEMO1)
TST_MEM_ROW1         STRING(60)
TST_MEM_ROW2         STRING(60)
TST_MEM_ROW3         STRING(60)
TST_MEM_ROW4         STRING(60)
TST_MEM_ROW5         STRING(60)
                  .
```

Figure 2.

By using entry fields in the screen structure that USE each variable in the previous group, you can enter data into the memo. The screen structure could look like Figure 3.

```
SCREEN      SCREEN          PRE(SCR),WINDOW(12,74),HUE(15,4)
               ROW(1,1)     STRING('┌─{72}┐'),HUE(15,4)
               ROW(2,1)     REPEAT(10);STRING('│<0{72}>│'),HUE(15,4) .
               ROW(12,1)    STRING('└─{72}┘'),HUE(15,4)
               ROW(2,31)    STRING('UPDATE TSTMEMO1')
               ROW(4,4)     STRING('ACCOUNT:'),HUE(7,4)
               ROW(5,4)     STRING('NAME    :'),HUE(7,4)
               ROW(6,4)     STRING('MEMO1   :'),HUE(7,4)
MESSAGE        ROW(3,23)    STRING(30),HUE(15,4)
                            ENTRY,USE(?FIRST_FIELD)
               ROW(4,13)    ENTRY(@N_4),USE(TST:ACCOUNT),HUE(15,4),SEL(0,7)
               ROW(5,13)    ENTRY(@s30),USE(TST:NAME),HUE(15,4),SEL(0,7)
               ROW(6,13)    ENTRY(@s60),USE(TST_MEM_ROW1),LFT,HUE(15,4)
               ROW(7,13)    ENTRY(@s60),USE(TST_MEM_ROW2),LFT,HUE(15,4)
               ROW(8,13)    ENTRY(@s60),USE(TST_MEM_ROW3),LFT,HUE(15,4)
               ROW(9,13)    ENTRY(@s60),USE(TST_MEM_ROW4),LFT,HUE(15,4)
               ROW(10,13)   ENTRY(@s60),USE(TST_MEM_ROW5),LFT,HUE(15,4)
                            ENTRY,USE(?LAST_FIELD)
                            PAUSE(''),USE(?DELETE_FIELD)
            .
```

Figure 3.

Unfortunately, when you use the previous technique (with entry fields), you lose the basic word processing-type control over the memo. The previous example references the memo in 60 character strings and treats each string like a separate field. This is one way of using groups to manipulate memos. Another way to use groups is to make a memo for a file actually look like multiple memos. We will use the previous file definition with a different size memo and different group structure to illustrate this technique. The file structure and group structure would now look like Figure 4.

```
TSTMEMO1    FILE,PRE(TST),CREATE,RECLAIM
BY_ACCOUNT     KEY(TST:ACCOUNT),NOCASE,OPT
BY_NAME        KEY(TST:NAME),DUP,NOCASE,OPT
MEMO1          MEMO(540)
RECORD         RECORD
ACCOUNT        SHORT
NAME           STRING(30)
            . .

            GROUP,OVER(TST:MEMO1)
TST_MEM_ROW1   STRING(180)
TST_MEM_ROW2   STRING(180)
TST_MEM_ROW3   STRING(180)
            .
```

Figure 4.

The screen structure would now use the variables in the previous group structure with TEXT fields to emulate the three memos. The screen structure could look like Figure 5.

```
SCREEN   SCREEN          WINDOW(18,74),PRE(SCR),HUE(15,4)
            ROW(1,1)     STRING('┌─(72)─┐'),HUE(15,4)
            ROW(2,1)     REPEAT(16);STRING('│<0(72)>│'),HUE(15,4) .
            ROW(18,1)    STRING('└─(72)─┘'),HUE(15,4)
            ROW(2,31)    STRING('UPDATE TSTMEMO1')
            ROW(7,4)     STRING('MEMO1  :'),HUE(7,4)
            ROW(11,4)    STRING('MEMO2  :'),HUE(7,4)
            ROW(15,4)    STRING('MEMO3  : '),HUE(7,4)
MESSAGE     ROW(3,23)    STRING(30),HUE(15,4)
            COL(53)      ENTRY,USE(?FIRST_FIELD)
            ROW(4,4)     STRING('ACCOUNT:'),HUE(7,4)
            COL(13)      ENTRY(@N_4),USE(TST:ACCOUNT),HUE(15,4),SEL(0,7)
            ROW(5,4)     STRING('NAME   :'),HUE(7,4)
            COL(13)      ENTRY(@S30),USE(TST:NAME),HUE(15,4),SEL(0,7)
            ROW(7,13)    TEXT(3,60),USE(TST_MEM_ROW1),LFT
            ROW(11,13)   TEXT(3,60),USE(TST_MEM_ROW2),LFT
            ROW(15,13)   TEXT(3,60),USE(TST_MEM_ROW3),LFT
            ROW(10,73)   ENTRY,USE(?LAST_FIELD)
            COL(73)      PAUSE(''),USE(?DELETE_FIELD)
```

Figure 5.

Now, as far as the user is concerned, there are three separate memos on the screen.

Note: If you have defined a memo's capacity for data entry with a certain row and column configuration, and data has been entered into the memo using that configuration, the modification of the rows and columns configuration on the entry screen will alter the appearance of the data in the memo when re-displayed for update. The reason for this is that when each row is adjusted for word wrapping, spaces are added to the end of the row. Since a soft return is not at the end of the row, when the memo is loaded into the new row and column configuration, there is no way to know where the current row ends and the next row begins.

Printing Memos

For an example of how to print memos, we will use the file structure in Figure 2. The memo in that file has a length of 300. Let's assume that the data was entered into this memo using a TEXT(5,60) screen field. This is important because if the TEXT field was TEXT(6,50) we would define the variable that redefines the memo differently. To make the printed output look exactly as it did when it was entered into the memo through the TEXT field, we define a variable with the same dimensions as the TEXT field, as in Figure 6.

```
MEMO_DETAIL       STRING(60),DIM(5),OVER(TST:MEMO1)
```

Figure 6.

The report structure in Figure 7 defaults to printing to LPT1. The MEMLIN variable is where each row of the memo will be loaded to print. The DETCTL variable will hold the two control characters for carriage return and line feed.

```
REPORT   REPORT
DETAIL      DETAIL
MEMLIN         STRING(80)
DETCTL         STRING(2)
```

Figure 7.

The code used to print the memo is shown in Figure 8:

```
LOOP UNTIL EOF(TSTMEMO1)              ! BEGIN LOOP TO READ TSTMEMO1 FILE
      NEXT(TSTMEMO1)                  ! RETRIEVE RECORD FROM FILE
      DETCTL = '<13,10>'              ! LOAD CARRIAGE RETURN/LINE FEED
      LOOP X# = 1 TO 5 BY 1           ! BEGIN LOOP TO PRINT EACH MEMO
         MEMLIN = MEMO_DETAIL[X#]     ! LOAD X# ROW OF MEMO INTO MEMLIN
         PRINT(DETAIL)                ! PRINT MEMO ROW THAT WAS LOADED
      .
      MEMLIN = ALL(' ',LEN(MEMLIN))   ! LOAD SPACES INTO MEMLIN
      DETCTL = ' <12>'                ! LOAD FORM FEED INTO CONTROL VARIABLE
      PRINT(DETAIL)                   ! SET PRINTER TO NEXT PAGE
.
```

Figure 8.

Obviously, this is not all the code required to print the memo, but it is an example of the main logic to accomplish our task. The previous example could also be output to a file or different device by using the DEVICE attribute on the report structure.

Writing Memos to a DOS File

Clarion supports three types of DOS files: binary, comma-delimited, and ASCII files. A full explanation of the different types of DOS files can be found in the Language Reference manual under DOS files. In this example, we will show you how to write memos to comma-delimited and binary DOS files. For a DOS ASCII file, just change the COMMA attribute on the DOS structure to ASCII. For the first example, we will again use the file in Figure 2. The DOS file structure for our comma-delimited example can be seen in Figure 9.

```
DOSFIL   DOS,PRE(DOS),COMMA
RECORD      RECORD
DOSGRP         GROUP
MEMDET1           STRING(60)
MEMDET2           STRING(60)
MEMDET3           STRING(60)
MEMDET4           STRING(60)
MEMDET5           STRING(60)
          . . .
```

Figure 9.

Let's again assume that the data was entered into this memo using a TEXT(5,60) screen field. In the comma-delimited format, each field defined within the group in Figure 9 will be surrounded by quotes and comma-delimited. This seems to go against the rules of groups because they are treated like one continuous character field, but that is how comma-delimited files work when using groups in Clarion. You will notice that there is a field in the group for each row in the memo. This is done to keep the format of the memo the same. Remember that because the data was entered using a TEXT(5,60) screen field, the only way to keep that format is to reference the memo by each row.

An example of the main logic used to load the comma-delimited file could be:

```
CREATE(TSTDOS1)              ! CREATE DOS FILE
OPEN(TSTMEMO1)               ! OPEN CLARION FILE
SET(TSTMEMO1)                ! START AT THE BEGINNING OF THE FILE
LOOP UNTIL EOF(TSTMEMO1)     ! LOOP UNTIL END OF TSTMEMO1 FILE
   NEXT(TSTMEMO1)            ! GET THE NEXT RECORD
   DOS:DOSGRP = TST:MEMO1    ! SET DOS FILE GROUP = CLARION MEMO
   ADD(TSTDOS1)              ! ADD RECORD TO DOS FILE
.
```

For the second example of writing memos to a binary DOS file, we will use the file in Figure 2 again. The DOS file structure for our binary file can be seen in Figure 10.

```
DOSFIL      DOS,PRE(DOS)
RECORD         RECORD
DOSGRP            GROUP
MEMDET               STRING(60),DIM(5)
                . . .
```

Figure 10.

We will use the same guidelines as in our first example. The binary file will contain only the data that was in our memos and each binary record will be 300 characters long. The code for this example is the same as the comma-delimited example, except that the DOS file structure (by not specifying any format) defaults to binary format. As you can see, there is a difference in the way the group structure is defined in the two previous DOS file structures. This is because you cannot use the DIM( ) dimension attribute in a comma-delimited DOS file structure in Clarion.

Appending Memos

For this example of manipulating memos, let's assume we have two data files and each file contains memos. We will make a third data file that contains memos and create the memos for the third file by combining each of the memos from our first two files. Figure 11 illustrates our first two files.

```
FILE1       FILE,PRE(FI1),CREATE,RECLAIM
BY_ACCOUNT     KEY(FI1:ACCOUNT),NOCASE,OPT
BY_NAME        KEY(FI1:NAME),DUP,NOCASE,OPT
MEMO1          MEMO(300)
RECORD         RECORD
ACCOUNT           SHORT
NAME              STRING(30)
               . .
FILE2       FILE,PRE(FI2),CREATE,RECLAIM
BY_ACCOUNT     KEY(FI2:ACCOUNT),NOCASE,OPT
BY_NAME        KEY(FI2:NAME),DUP,NOCASE,OPT
MEMO1          MEMO(200)
RECORD         RECORD
ACCOUNT           SHORT
NAME              STRING(30)
               . .
```

Figure 11.

You will notice that the size of the memo is different for the two files. We have done this to illustrate a point about referencing memos by rows. In order to keep the format of the memos when they were entered into the two files, we will need to find out what the row and column combinations were for both of our files. For FILE1, the memos were entered using a TEXT field of 5 rows by 60 columns. For FILE2, the memos were entered using a TEXT field of 4 rows by 50 columns. To determine the best size for our memo in our third file, we will take the larger of the two column sizes and multiply that by the total number of rows for the two memos (60 x (5 + 4)). We now know that we need a memo size of 540. The row and column configuration for our new memo is 9 rows by 60 columns. When the FILE2 memo is appended to the FILE1 memo, the rows of the FILE2 memo will be padded with spaces in the new FILE3 memo in order to keep the original format of the FILE2 memo. The new file could look like the one shown in Figure 12.

```
FILE3       FILE,PRE(FI3),CREATE,RECLAIM
BY_ACCOUNT     KEY(FI3:ACCOUNT),NOCASE,OPT
BY_NAME        KEY(FI3:NAME),DUP,NOCASE,OPT
MEMO1          MEMO(540)
RECORD         RECORD
ACCOUNT           SHORT
NAME              STRING(30)
   . .
```

Figure 12.

We now need to create three group structures for our files so that we can reference each of the memos by their respective rows, as shown in Figure 13.

```
          GROUP,PRE(GP1),OVER(FI1:MEMO1)
WORKMEM   STRING(60),DIM(5)
          .

          GROUP,PRE(GP2),OVER(FI2:MEMO1)
WORKMEM   STRING(50),DIM(4)
          .

          GROUP,PRE(GP3),OVER(FI3:MEMO1)
WORKMEM   STRING(60),DIM(9)
          .
```

Figure 13.

The code for this example could be as follows:

```
OPEN(FILE1)                                  ! OPEN FILE1
OPEN(FILE2)                                  ! OPEN FILE2
CREATE(FILE3)                                ! CREATE FILE3
SET(FI1:BY_ACCOUNT)                          ! SET TO BEGINNING OF FILE1
LOOP UNTIL EOF(FILE1)                        ! LOOP UNTIL END OF FILE1
   NEXT(FILE1)                               ! GET NEXT RECORD FOR FILE1
   FI3:RECORD = FI1:RECORD                   ! LOAD RECORD INFO FROM FILE1
   LOADREST# = 0
   Z# = 1                                    ! SET FILE3 INDEX TO ROW TO 1
   LOOP A# = 5 TO 1 BY -1                    ! LOOP THRU ROWS OF FI1:MEMO1
      IF GP1:WORKMEM[A#] > ' ' OR ¦          ! IF ROW HAS DATA IN IT OR
         LOADREST#                           ! LOAD REST OF ROWS
            GP3:WORKMEM[A#] = GP1:WORKMEM[A#]
            IF ~LOADREST#
               LOADREST# = 1
               Z# = A# + 1                   ! SET ROW FOR FILE2 MEMO LOAD
            .
      .
   .
   FI2:ACOUNT = FI1:ACCOUNT                  ! SET FILE2 ACCOUNT FOR GET
   GET(FILE2,FI2:BY_ACCOUNT)                 ! GET MATCHING ACCOUNT
   IF ~ERRORCODE()                           ! IF NO ERROR LOAD FILE2 MEMO
      LOOP B# = 1 TO 4 BY 1                  ! LOOP THRU ROWS OF FI2:MEMO1
         IF GP2:WORKMEM[B#] > ' '            ! IF ROW HAS DATA IN IT
            GP3:WORKMEM[Z#] = GP2:WORKMEM[B#]
            Z# += 1                          ! BUMP INDEX INTO FILE3 MEMO
         .
      .
   .
   ADD(FILE3)                                ! ADD NEW RECORD TO FILE3
.
```

Figure 14.

While this is not all of the required code to accomplish this task, it is an example of the main logic.

Searching a memo with INSTRING

You can use INSTRING to search through memos to find a substring. The second parameter in the Reference Manual indicates that you can only pass a string to INSTRING. But you can pass a direct reference to a memo or even a group that contains multiple string(255) references in it. Try it!