C for yourself

part 1

Norman Rokke

When C99 first became available at the user group I got a copy with the intention of learning about C. However, when I couldn't get a running program from the sample programs, I put it aside and concentrated on other programming interests. Some years later I took a course in C but I never returned to C99. Recently I had an idea for a program I wanted to write and began writing it in Extended BASIC. I got a portion of the program written but it became obvious that Extended BASIC was not the most appropriate language.

One of the problems was speed. Because of this and other deficiencies of Extended BASIC which I won't go into now, I decided to take another look at C99. Again, when I tried to get running programs from some of the sample programs, I experienced difficulties. This time however I finally managed to figure out what I was doing wrong and was able to get the sample programs to run. I have been able to code a considerable portion of my program in C99 and am very pleased with the language.

In this article I want to explain how C99 can be used to create a simple program, just to show the overall process from start to finish. This process consists of four steps.

1. Using an editor to create the C99 program.

2. Using the C99 compiler to compile the C99 program and produce assembly language source code.

3. Using the assembler to produce executable code (EA option 3).

4. Creating EA option 5 executable code.

Let's look at a simple C99 program that prints **Hello world** on the screen. I've read that one of the developers of C wrote a program to do this as the first program ever written in the language.

As I describe how to carry out each step I'll assume you have the set of C99 disks offered by Vern Jensen in Micropendium. I'll also assume that you have a two disk drive system and that the Funnelweb disk is in drive 1 and that you will save your C99 programs and any other necessary files on drive 2.

To create a C99 program you use an editor. From the menu choose 1 PROGRAM ED. This editor is essentially the TI-Writer or Funnelweb editor modified so that it does not cause problems with the C99 compiler. I assume that you are familiar with the basic commands of the editor. Use the editor to type in the following program.

```
extern grf1(),printf();
main()
   {
   grf1();
   printf("Hello world");
   }
```

After you have typed the program in, save it with the file name DSK2.HELLO;C using SF. The ;C ending of the filename is a convention to make it easy to recognize this file as a file which contains C99 program code. Then leave the editor and go to the menu screen.

Before describing how the program is compiled, I'd like to briefly explain the above program. The first line

extern grf1(),printf();

provides the compiler with information it needs to properly handle the program you are writing. Much of C99 is contained in functions which are stored in library files. grf1() and printf() are two such functions. The code for providing the capabilities of these two functions is in separate files which are part of the C99 package. The grf1() function puts the computer in graphics mode (24 rows, 32 columns) rather than the default text mode (24 rows, 40 columns). The printf function works much like the PRINT statement of Extended BASIC. In order to use these functions in our program, we must tell the compiler that we are going to be using these functions which are outside of or *external* to our program. This statement also requires that at the appropriate time we must provide the necessary library files so that everything will work properly. We'll see later when and how that is done.

Every C99 program consists of one or more functions. These functions are like subprograms in Extended BASIC. The second line

main()

identifies the one function which **must** be present in every C99 program, a function whose name is main. The parentheses after the name identify it as a function.

The other functions in a program are either library functions or functions created by the programmer. This program does not use any of the latter. The most important thing to know about the function main is that the program begins by executing the statements in this function.

The **{** on the next line marks the beginning of a block of code for the function main.

The next line

grf1();

sets the computer in graphics mode. Note that statements in C99 end with a semicolon. This applies to the **extern** declaration statement above also.

The next line

printf("Hello world.");

uses the printf function to print something on the screen.

The **}** on the last line marks the end of the block of code for the function main.

Now let's compile the program we've written. Choose 4 C-COMPILER from the menu. Press ENTER to accept the default values for each of the three questions. Next press ENTER to accept HELLO;C as the input file. Change the name of the output file to DSK2.HELLO;S and press ENTER. The ;S ending of the file name is a convention to help you recognize this file as an assembly language source file. Now press FCTN-6 to start the compilation process. Type N in response to the question of whether you want to compile again.

If the compilation completes and the message indicates zero errors, you are ready to go on to the next step. However, if there are errors, you must go back to step one and edit the C99 program to correct the errors. If errors are reported use PROGRAM ED and make sure your program is identical to the one above. Then save the file and try the compilation again. If your file is exactly like the program above it will compile successfully.

The third step involves assembling the code produced by the compiler. Choose 2 ASSEMBLER from the main menu. Press ENTER twice to accept DSK2.HELLO;S as the source file and DSK2.HELLO;O as the object file to be created. The ;O ending is a convention to help you recognize the file as an Editor Assembler option 3 object code file. You can press ENTER twice more to accept the default values for the last two prompts also. Press FCTN-6 to begin assembly. The assembly should go without a hitch and you should get a report of no errors. Press ENTER to go back to the main menu.

Finally we have a program that we can run. To be more precise, it will run if we load the additional files which are necessary. Every C99 program needs the file CSUP in order to run. This file contains code which sets up the default text mode before the program runs. It also contains code to cleanly end the

program when the program is finished. Copy the file CSUP from the C99-LIBS disk onto the disk containing the program files we have been working with.

Since we used library functions in our program, we also need to supply the files that contain those library functions. the function grf1() in is the library file GRF1 and printf() is in the file PRINTF. Both of these files are on C99-LIBS. Copy both of them to the disk with the program files and CSUP.

Now we are ready to run the program. Choose 3 LOADERS from the menu. Then choose 4 LOAD/RUN (E/A). Enter DSK2.HELLO;O as the filename and press ENTER. When the cursor returns press FCTN-3 and enter the filename DSK2.CSUP and press ENTER. When the cursor returns press FCTN-3 and enter the filename DSK2.GRF1 and press ENTER. When the cursor returns press FCTN-3 and enter the filename DSK2.PRINTF and press ENTER. When the cursor returns press FCTN-3 and press ENTER.

You will see a screen filled with DEF Table entries. Use FCTN-D to place the cursor on the **S** of **START**. Then press FCTN-6 and the program will run. Although the program doesn't do anything impressive it should be satisfying to see that what you have written actually produces a result. As far as I know there is no way back to the main menu from here so press FCTN-=.

As satisfying as it may be to get the program running. I'm sure you will agree that there has to be an easier way to run the program than what we just went through. There is.! What we must do is create an E/A option 5 program. In order to do this we need three more files on our program disk. Copy the files C99PFI, C99PFF, and SAVE from the C99-LIBS disk to your program disk.

C99PFI contains the DEFs SLOAD and SFIRST which are needed by the E/A SAVE utility. C99PFF contains the SLAST DEF also needed by SAVE. SAVE is the E/A utility program used for creating program image (E/A option 5) files.

Choose 3 LOADERS from the main menu and then choose 4 LOAD/RUN (E/A). The order in which the files are entered is critical so enter them just as listed below.

Enter filename DSK2.C99PFI and press ENTER. When the cursor returns press FCTN-3 and enter filename DSK2.HELLO;O and press ENTER. When the cursor returns press FCTN-3 and enter filename DSK2.CSUP and press ENTER. When the cursor returns press FCTN-3 and enter filename DSK2.GRF1 and press ENTER. When the cursor returns press FCTN-3 and enter filename DSK2.PRINTF and press ENTER. When the cursor returns press FCTN-3 and enter filename DSK2.C99PFF and press ENTER. When the cursor returns press FCTN-3 and enter filename DSK2.SAVE and press ENTER. When the cursor returns press FCTN-3 and press ENTER.

You will now see a screen filled with DEF Table entries. The entry we want is SAVE and it is not on this screen. Press ENTER to get another screen of DEF Table entries. SAVE isn't on this screen either. Press ENTER to get to the third screen of DEF Table entries. SAVE is the last entry. Use FCTN-D to position the cursor over the S of SAVE. Then press FCTN-6.

At the prompt for the filename enter DSK2.HELLO. Then press ENTER and let the SAVE utility do its work.

When the program finishes press FCTN-9 to go back to the LOADERS menu. Select 3 PROGRAM (E/A). Enter DSK2.HELLO and press ENTER. You now have your simple C99 program in a much more easily run-able form.

I use the E/A option 3 program to test for errors I don't make a program image file until the program is rather well debugged.

C for yourself

part 2

Norman Rokke

First, there is an easier way to run an option 3 program or create an option 5 program. It's amazing what you can learn when you read the documentation. This method uses the CLOAD program which is on the Funnelweb disk and is accessible through option 5 C-LOADER on the main menu. This program lets you make a list of option 3 files in a DV80 file. It reads that file and loads all of the option 3 files in the list. Then it lets you choose the entry point to run the program.

Let's look at how we would use this loader. Use the editor to create a file with the following contents.

DSK2.HELLO;O
DSK2.CSUP
DSK2.GRF1
DSK2.PRINTF

Save this file as DSK2.HELLO;3 and leave the editor. I use ;3 to indicate that this is a file with a list of files to be used for an option 3 program. Select 5 C-LOADER from the menu. Enter DSK2.HELLO;3 and press ENTER. After the files have been loaded, press FCTN-3 to clear the entry and press ENTER. When you get the screen of DEF Table entries, move the cursor so it is over the S of START and press FCTN-6.

Now, no matter how much you may change HELLO;C, you can run the program just this easily whenever necessary.

The loader can also be used to simplify creating an option 5 program. For our sample program this would require a DV80 file with the following contents.

DSK2.C99PFI
DSK2.HELLO;O
DSK2.CSUP
DSK2.GRF1
DSK2.PRINTF
DSK2.C99PFF
DSK2.SAVE

Save this file as DSK2.HELLO;5. I use the ;5 ending to remember that this file contains a list of files for creating a program image file of the program. It is used just like the option 3 file above except that you select the DEF SAVE instead of START. Remember, SAVE is on the third screen. Press ENTER twice to get to that third screen.

In my last article I mentioned that one advantage of C99 was speed. Another of its advantages is that you have 256 characters (32 sets) available for use just as in pure assembly. This is twice as many as in BASIC and more than twice as many as in Extended BASIC. No more having to define a character for one purpose in one part of the program only to redefine it for some other use later in the program.

In Extended BASIC I have at times used redefined characters to center text which is an odd number of characters long. However, I had to be very selective about when I did this because of the limited number of characters available. In C99, this technique can be used much more freely because of the large number of characters available.

I'd like to consider a C99 program to center some odd length text on the screen. Let's use the text ODD. In order to center this we need to know the hex character definitions for the letters O and D. Using the CALL CHARPAT statement of XB we can find that these are as follows.

O "007C44444444447C"
D "0078242424242478"

These hex characters define how the text character looks on the screen. The hex characters define the left and right halves of the text character as shown below for the letter O.

0	0
7	C
4	4
4	4
4	4
4	4
4	4
7	C

The patterns for the entire word ODD would look like this.

0	0	0	0	0	0
7	C	7	8	7	8
4	4	2	4	2	4
4	4	2	4	2	4
4	4	2	4	2	4
4	4	2	4	2	4
4	4	2	4	2	4
7	C	7	8	7	8

What we need to do to center the text is to create four text characters which have the above definitions as their middle six columns and with zeros in the columns on either end as shown below.

0	0	0	0	0	0	0	0
0	7	C	7	8	7	8	0
0	4	4	2	4	2	4	0
0	4	4	2	4	2	4	0
0	4	4	2	4	2	4	0
0	4	4	2	4	2	4	0
0	4	4	2	4	2	4	0
0	7	C	7	8	7	8	0

This gives us definitions for 4 text characters which when placed on the screen will display the text ODD but centered within the four character string. These character definition strings are

"0007040404040407"
'00C74242424242C7"
'0087424242424287" and
'0080404040404080".

Now we can write a C99 program to redefine some characters and print the result on the screen.

```
extern grf1(),printf(),chrdef();

main()
  {
  grf1();
  chrdef(97,"0007040404040407");
  chrdef(98,"00C74242424242C7");
```

```
chrdef(99,"0087424242424287");
chrdef(100,"0080404040404080");
locate(1,15);
printf("abcd");
locate(2,15);
printf("EVEN");
locate(23,1);
}
```

We begin by specifying those functions which are external to our code. The only new thing here is chrdef(). This is a C99 function contained in GRF1 which performs the same job as CALL CHAR in XB.

In the program itself we begin by putting the computer in graphics mode. Then we redefine the four characters with ASCII codes 97 through 100. These are the letters a,b,c, and d.

The locate function allows us to place the cursor at a particular row and column on the screen. This function is part of CSUP and therefore does not need to be included in an extern statement. We then print the string of redefined characters. Then we print a even numbered string of characters beneath it for comparison.

C for yourself

part 3

Norman Rokke

Now we will look at writing a C99 program for providing character definitions for centered text of odd length. Before getting to the code for doing this, let's consider how this can be done.

Let's use "ODD", the same text string that we used last time. We saw that the patterns for these three text characters would look like this.

0	0	0	0	0	0
7	C	7	8	7	8
4	4	2	4	2	4
4	4	2	4	2	4
4	4	2	4	2	4
4	4	2	4	2	4
4	4	2	4	2	4
7	C	7	8	7	8

We also saw that the patterns for the four characters which we wanted to produce would look like this.

0	0	0	0	0	0	0	0
0	7	C	7	8	7	8	0
0	4	4	2	4	2	4	0
0	4	4	2	4	2	4	0
0	4	4	2	4	2	4	0
0	4	4	2	4	2	4	0
0	4	4	2	4	2	4	0
0	7	C	7	8	7	8	0

Let's think of the first collection of patterns as a string with 48 hex characters numbered from 0 to 47. The reason for starting at 0 will be explained a bit later.

0	1	16	17	32	33
2	3	18	19	34	35
4	5	20	21	36	37
6	7	22	23	38	39
8	9	24	25	40	41
10	11	26	27	42	43
12	13	28	29	44	45
14	15	30	31	46	47

Likewise, the collection of patterns for the redefined characters can be thought of as a string of 64 hex characters numbered from 0 to 63.

0	1	16	17	32	33	48	49
2	3	18	19	34	35	50	51
4	5	20	21	36	37	52	53
6	7	22	23	38	39	54	55
8	9	24	25	40	41	56	57
10	11	26	27	42	43	58	59
12	13	28	29	44	45	60	61
14	15	30	31	46	47	62	63

If you look carefully you can see that the hex characters which are in even numbered positions in the first grid are in odd numbered positions in the second grid. More specifically, each hex character in an even position of our original string is moved forward 1 position. The character at position 0 winds up at 1, the character at 2 winds up at 3, the character at 16 winds up at 17 and so on.

What happens to the characters at odd positions? Well, the character at 1 winds up at 16, the character at 3 winds up at 18 and so on. Each character in an odd position in the original string winds up at a position 15 higher in the second string.

One last detail must be noted. The first 8 even positions of the second string must be assigned the hex character 0 as must the last 8 odd positions in this string.

This gives us the method for doing the conversion. Starting with the original text string, get the hex string for each text character in order, and join all of these together into one long hex character string.

Then, create a new hex string by moving the characters at even positions in the original string one position forward in the new string. Move characters in odd positions 15 positions forward in the new string. Fill the first 8 even positions of the new hex string with character 0 and do the same with the last 8 odd positions.

Then break up the new hex string up into 16 character strings to be used for redefining characters.

Now that we have a method for accomplishing our task, we can turn our attention to the code. When we do that we encounter a problem. In C99 there is not an available function which does what CALL CHARPAT does in XB. Well I guess that does it for this column.

Wait a minute. I could run an XB program and print out each of the character definition strings obtained from CALL CHARPAT and type them in this article. Well, I could. But I'd probably make at least one mistake, and you might also make a mistake when typing from this article. That doesn't sound like a very good solution.

Hey! I just had another idea. We can get the character definition strings from XB. The C99 program that needs them is a DV80 file. We can produce a DV80 file from an XB program. Let's just write an XB program to produce a DV80 file with the C99 code needed to bring in the character definitions.

Below is an XB program that will do this. Type this program in and run it. Notice that the C99 program file will be produced on DSK2. If you want produce it on another drive, change line 100. I'll discuss how the C99 code works a little later.

```
100 OPEN #1:"DSK2.CENTODD2;C
",OUTPUT,DISPLAY  ,VARIABLE 8
0
110 CH=32
120 CALL CHARPAT(CH,A$)
130 CALL CHARPAT(CH+1,B$)
140 CALL CHARPAT(CH+2,C$)
150 CALL CHARPAT(CH+3,D$)
160 PRINT #1:"  strcpy(cd,"&
CHR$(34)&A$&B$&C$&D$&CHR$(34
)&");"
170 FOR CH = 36 TO 124 STEP 4
180 CALL CHARPAT(CH,A$)
190 CALL CHARPAT(CH+1,B$)
200 CALL CHARPAT(CH+2,C$)
210 CALL CHARPAT(CH+3,D$)
220 PRINT #1:"  strcat(cd,"&
CHR$(34)&A$&B$&C$&D$&CHR$(34
)&");"
230 NEXT CH
240 CLOSE #1
```

Before we look at the rest of the C99 program, you may want to copy to the program disk the other files that will be needed. These include CSUP, GRF1, and PRINTF from the libraries disk. You will also need STRINGFNS from that disk. From the utilities disk you will need SEGSTR/O.

Below is the C program. Load the file CENTODD2;C into the editor and add the additional lines to what was produced by the XB program. The bold text is the code to be added.

```
#define NULL 0
extern grf1(),printf(),chrdef(),hchar(),segstr();

main()
  {
  int length,i,j,k,offset;
  char cd[1537];
  char text1[32],text2[33],hex1[513];
  char hex2[529],str[17];

  grf1();

  strcpy(cd,"0000000000...
  ...
  ...
  ...
```

```
 strcat(cd,"0010101000...

 strcpy(text1,"ODD");
 length = 3;
 i=0;
 for(j=0;j<length,j=j+1)
  {
  offset=16*(text1[j]-32);
  for(k=0;k<16;k=k+1)
   {
   hex1[i]=cd[offset+k];
   i=i+1;
   }
  }
 hex1[i]=NULL;
 hex2[i+16]=NULL;
 for(i=0;i<16*length;i=i+1)
  {
  hex2[i+1]=hex1[i];
  i=i+1;
  hex2[i+15]=hex1[i];
  }
 for(i=0;i<16;i=i+2)
  hex2[i]='0';
 for(i=length*16+1;i<16*(length+1);i=i+2)
  hex2[i]='0';
 strcpy(text2,"chrdef(nnn,\"");
 for(i=0;i<=length;i=i+1)
  {
  segstr(hex2,str,16*i,16);
  chrdef(128+i,str);
  text2[12]=NULL;
  strcat(text2,str);
  strcat(text2,"\");");
  locate(i+3,1);
  printf(text2);
  }
 for(i=0;i<=length;i=i+1)
  hchar(1,15+i,128+i,1);
 locate(2,15);
 printf("EVEN");
 locate(23,1);
  }

#include "DSK2.STRNGFNS"
```

After saving the program you may want to create the file for the C-Loader. This file would contain the following.

DSK2.CENTODD2;O
DSK2.CSUP
DSK2.GRF1
DSK2.PRINTF
DSK2.SEGSTR/O

When you compile this program you need to choose y for the third option Assume long jump for it to compile and assemble properly.

Now let's look at the code. The first line provides information to the compiler. It instructs the compiler to replace the text string NULL with the value which follows in the #define directive, namely '\0'. This value represents the character whose ASCII code is zero.. In C, individual characters are designated by enclosure between single quotes. The backslash followed by a number is used for those characters which can not be readily typed from the keyboard.

We are going to use the ASCII zero character in our program and NULL is also used in the code in file STRINGFNS which we will use.

In the extern statement, we see some new functions. The segstr() function is from Bruce Harrison's utility programs and provides the same capability as SEG$ in XB. The function hchar() works like CALL HCHAR in XB. Both of these are in GRF1.

The first four lines in main() are variable declarations. In C there is only one rule for naming variables. The name must start with a letter and the remaining characters may be either letters or digits. You can not tell the type of data stored in a variable based on its name as you can in XB. You must associate each of your variables with one of the available data type.

In C99 the available data types are integer, character and pointer. Integer variables can contain any whole number in the range from -32768 to 32767.A character variable can contain any individual character (letter, digit, punctuation etc.) with ASCII value from 0 to 255. We will not discuss pointer variables at this time.

You may also work with arrays of integers or characters. In C99 arrays are limited to one or two dimensions.

In our program we are declaring 5 integer variables. All of these are listed in the same type declaration. We also need several character array variables. In C character arrays are used to represent string values.

In C strings are terminated by an ASCII zero character. When setting the size of a character array to be used for string data, you must remember to leave room for the ASCII zero. The variable cd is a character array we will use to store the hex string for defining all of the characters from 32 to 127. This requires 16*96 or 1536 characters. We must set the size of the array at 1537 to allow for the ASCII zero. The variable text1 will be used to store the string of text characters to be centered (an odd number of characters with maximum of 31). We will use hex1 to store the hex definition strings of the characters in text1 (maximum 512 characters). In hex2 we will store we will store the hex definitions of the redefined characters. The hex definition of a single character will be temporarily stored in str. Variable text2 will be used to hold and print on screen the C99 character definition statements needed for the redefined characters.

When using arrays in C it is important to remember that the first position in the array always has index 0. This is why we started numbering the cells in the grids at zero. For example str would have valid indexes in the range from 0 to 16. The number inside brackets in the declaration is the number of values that can be stored.

It is also important that you realize that the compiler does not check to see that array indexes are in the proper range. For example a reference to str[37] would not prevent your program from compiling and assembling. However, it probably would cause some problem somewhere and worst of all the problem

would show up as something totally unrelated to the source of the problem making it very difficult to debug.

C puts a lot of responsibility on the programmer to handle details that can be ignored in XB. The advantages of the language don't come without cost.

We begin by putting the computer in graphics mode. Next we come to the code produced by the XB program. This code creates the string cd. The function strcpy() performs the same task that CD$="ABC" would do in XB. It lets us assign a value to a string variable. It provides the ASCII 0 for string termination also. The strcat() function does what CD$ = CD$&"DEF" would do in XB. It also takes care of the ASCII 0 terminator. The result is that the string cd contains the hex character definitions of all of the characters from 32 to 127.

Next we assign "ODD" to text1 and set length to 3. Note that assignment of a value to an integer variable is done just as in XB.

Now we need to get the character definitions for each of the characters in text1. This is done by a nested pair of for loops. We use i as an index to the position in the hex string hex1. This is set to zero before the loops.

The for statement is followed by three expressions separated by semicolons. The first expression gives an initial value to the loop control variable. The second expression is used to determine when to stop the loop. As long as this expression is true, the loop continues. When it becomes false, the loop ends. The last expression is performed at the end of each pass through the loop. In this case the loop starts with j=0 and continues for all values of j up to length-1. These we use as index positions to all of the characters in text1 which actually have data (0 to 2). The last expression takes care of increasing j by one each time through the loop.

For those of you who may know some C, I know that the incrementating of the loop control variable is not typically done like I have done it here. However, I wanted to keep things as simple as possible at the beginning. I will discuss the more typical way of doing this in a latter article.

Variable j is an index to a position in the string text1. The outer loop takes us through each of the positions in the string in text1.

Character data is stored as ASCII code values so text1[j] will be the ASCII code of one of the characters in the string we want to center. If j is 0, text[j] will be 79 (for 'O'). By subtracting 32 (ASCII code for the space character), we determine how many characters come before the one we are dealing with. Multiplying by 16 gives us the position in cd where the definition of the current character begins.

The inner loop runs from k=0 to k=15 to get each of the 16 hex characters that define the current text character. Inside the loop we also increment i.

Also notice the use of curly brackets to create a block of statements , all of which are done each time through the loop. Indentation also is used to make it easier to tell what is done inside the loop.

Next we put the ASCII zero character into hex1 and hex2. Variable hex2 is 16 characters longer because it defines one more text character than hex1. Since we are not creating these strings by using string functions, we are responsible for making sure the ASCII zero is in the proper place.

The next loop moves each character in hex1 to its proper position in hex2 as we discussed earlier. Characters at even indexes are moved ahead 1 position and characters at odd indexes are moved ahead 15 positions in the new hex string (hex2).

The next loop fills the first 8 even index positions in hex2 with '0'. Then the next loop fills the last 8 odd index positions in hex2 with '0'. Note that this is the character '0' (the same as "0" in XB) and not '\0'.

Next we redefine characters and print the appropriate C99 statements to do this on the screen. Outside the loop we define the first part of the line of text to display on the screen. Note the \" in the string that is to be assigned to text2. This is how we can put the " character inside a quoted string. The \" becomes only one character. It's comparable to using double quotes in XB to get one quote character.

Inside the loop we use segstr() to get individual character definition strings from cd. Then we redefine a character. Next we put the ASCII zero at the end of text2. The first time through the loop it will already be there since we used strcpy() outside the loop. Inside the loop we're going to add to the string. After the first time through the loop the ASCII zero will not be in this position so we have to put it there so we can add to the same basic string each time through the loop.

Next we add to text2 the hex string character definition. Then we add "); to the end of the string. Using the locate() function which we saw last time we position the cursor at column 1 of a particular row and then print the chrdef statement to the screen.

In the next loop we display the four redefined characters centered on the first row of the screen. I have not been able to use printf to deal with characters above 127 so I used the hchar() function. This is a function included in GRF1 which works much like CALL HCHAR in XB. The only difference is that you must supply the fourth argument (number of times to repeat character) even if it is 1 as in this case.

Then we print a centered line of text containing an even number of characters for comparison. The last locate statement positions the cursor at the bottom of the screen for printing the exit rerun? message when the program ends.

The #include compiler directive tells the compiler to deal with the text in the file STRINGFNS as if it were right here in this program. The file STRINGFNS contains the code for strcpy() and strcat() as well as for other functions which we don't use. Since the code for strcpy() and strcat() will be compiled when we compile our program we do not need extern statements for these functions.

While this program works, it leaves much to be desired. We really need a program that will let us enter the text we want to center. Also the program would be improved if we could write the character redefinition statements directly to a DV80 file so that they could be used directly in a C99 program without copying from the screen and typing.

C For Yourself

Part 4

Norman Rokke

Here are some improvements that could be made to the program we have been working on. When I wrote that I hadn't yet written the programs to accomplish those improvements. The bad news is that I still haven't written the programs. The good news is that this failure is caused by lack of time to write the programs rather than being unable to make a program work.

Rather than skip a month I thought it might be useful to look a bit more closely at how C lets us use code which is somewhere other than in our own program. The term library functions is used to refer to this kind of code and the files which contain these library functions are called library files.

We have used a number of library functions in previous articles. Functions such as **locate**, **printf**, **chrdef**, **hchar**, **strcpy**, **strcat**, and **segstr** are examples. Although they are all library functions, there are some differences in how they are used.

Let's start by looking at the function **locate**. The code for this function is in the file CSUP. This file is a DF80 file which contains already assembled code. We have seen how we include this file in our program at the appropriate time. We have also seen that this file **must** be included in every C99 program we write.

Because CSUP must be used with every program that we write, the compiler knows about everything that is in CSUP. Therefore, we do not need to tell the compiler about **locate** by using an extern declaration because the compiler already knows about it.

The files PRINTF, GRF1, and SEGSTR/O also contain already assembled code for library functions that we can use. These files do not need to be used with every program that we write and so the compiler does not know anything about functions like **printf**, **chrdef**, **hchar**, or **segstr**. In order to use these functions in our programs we use the extern declaration to tell the compiler about these functions that are somewhere else outside our program.

In the example programs of previous articles I used extern declarations including only those library functions which would be used in the program. Another way to supply this information is to use what is called a header file. This is a file which contains extern declarations containing all of the library functions in a particular library file.

The file GRF1;H is an example of such a file. This file contains extern declarations listing all of the functions in GRF1 including many which we didn't use such as **screen**, **clear**, **vchar**, and many others. If we put the compiler directive

```
#include DSKn.GRF1;H
```

at the beginning of our program we can use any of the library functions in GRF1 with no worry. The #include directive tells the compiler to find the file GRF1;H on the specified drive and treat the contents of that file just as if they were typed in our program at the point of the #include directive.

I prefer not to use the header file. I like to be able to look at the program code and know what library functions are being used and I can do that when the extern declarations are present in the program. When the header file is used, you can't determine which functions are used unless you examine the code in detail. On the positive side the use of the header file simplifies things and prevents misspelling of function names which would lead to compilation errors. Ultimately, it comes down to a matter of personal preference.

Before we leave library functions which are in DF80 files, let's consider some advantages and disadvantages of these types of library functions. First the advantage of these functions is that we never have to wait for the code which performs these actions to be compiled or assembled. The code is ready to execute and we simply add the necessary library file(s) after we have compiled and assembled our own program.

The possible disadvantage of this method can best be explained by considering the GRF1 library file. As we have seen the functions **grf1**, **chrdef**, and **hchar** are included in this file. However there are also many other functions such as **screen**, **chrset**, **clear**, **key**, as well as numerous sprite functions in GRF1.

When we add the file GRF1 to our program we get the code for these functions as well even though we don't use them. This code takes up memory which is not available for code which we might actually need. This makes the program larger than it needs to be with no benefit. In a very large program we might not have enough memory for our program code because of the memory wasted in storing functions we don't use.

This disadvantage does not apply to all DF80 library files. The files PRINTF and SEGSTR/O each contain only one function (**printf** and **segstr** respectively) and the use of these functions results in no wasted memory.

Before continuing, I'd like to make it clear that I'm not knocking anyone associated with the creation of DF80 library files for C99. I'm simply trying to point out some pros and cons connected with use of these files.

I realize that there may have been some efficiency gained from different library functions in GRF1 using the same code. Also the prospect of having individual files for each of the functions in GRF1 (over 20 in number) is not appealing.

The other type of library file that we can use is a DV80 file which contains C99 code. This code can be added to our program by means of a #include directive. We used this method in Part 3 to add the functions **strcpy** and **strcat** to our program.

These code for these functions is in the file STRINGFNS. This code is C99 code. The #include directive causes the compiler to treat this code just as if it were part of our program file and compile it along with the rest of our program. This means we don't have to go to the effort of retyping this code in our program in order to use it. We simply tell the compiler where to find the code and it does the rest.

Since the code for these two functions is really part of our program we do not have to use extern. These functions are part of our program. The code just happens to be in a separate file.

Now let's look at the advantages and disadvantages of this type of library function. In this case we are not dealing with code that is ready to be executed. The code for these functions must be compiled and assembled along with the rest of our program. Our program will take longer to compile and assemble than it would if we had these functions in DF80 format.

Another possible problem when using these types of library functions is that they may contain symbolic constants such as NULL in the library that we used. When I first tried to use this library, everything went well when I compiled the program but an error occurred when I tried to assemble the program.

When I checked the line that produced the problem, the only thing that looked unusual was the label NULL. I knew that I hadn't used this in any of my code so I checked the file STRINGFNS. In order to use the library functions I needed to know that the symbolic constant needed to be defined with a #define directive. I also needed to know what value needed to be assigned to NULL to make things work properly. Without this knowledge the library functions would be unusable.

It would be very helpful if those who supply library functions containing symbolic constants would include a comment indicating the appropriate #define directive needed to use the library.

Yet another problem with this type of library function is that the library may contain functions that we do not need. That is true of STRINGFNS. We only need 2 of the several functions in this file. Not only do we get executable code which contains code that we never use, but we have to wait for this unused code to be compiled and assembled.

This points to the biggest advantage of this type of library function. We don't have to put up with this wastefulness. Rather than using the entire file STRINGFNS, we can create a file which contains only the code for the functions we need.

To do this load the file STRINGFNS into the editor. We want to save lines 89 to 124 as a separate file. We can do this as follows. Choose SF. Use FCTN 2 to insert 89 124 before the filename and change the filename to CAT+CPY. The line should look like this:

89 124 DSK2.CAT+CPY

When it does press ENTER to save the file.

We are not quite finished. The file CAT+CPY also contains the code for **stncat** which we do not need. Load in the file CAT+CPY and erase the lines starting with stncat(through the line above strcat(. Then save the file.

Now if we change the last line of CENTODD2;C to

#include "DSK2.CAT+CPY"

we will be using only the functions that we need. When you make this change the E/A option 3 program that results is 30 sectors. The file using STRINGFNS was 35 sectors long.

END