Defining and Referencing Variables and Constants in High Level Assembler

Part I

Authors:
Steve Hobson
Sharuff Morsa
smorsa@uk.ibm.com
steve_hobson@uk.ibm.com

Introduction

Assembler programs reference storage areas during program execution. The storage areas may be defined within the scope of the application program code or be located elsewhere in memory. High Level Assembler (**HLASM**) provides two assembler directives which an assembler application programmer uses to define these storage areas: **DC** (Define Constant) and **DS** (Define Storage). The key difference being DC will initialise the storage area, DS will not.

The two assembler instructions not only define storage areas, they provide a name (symbol) by which the storage area may be referenced by the program. The HLASM Reference Guide refers to this as: '...Provides labels for these areas...'.

All storages areas addressed by your program are simply a series of binary ones and zeros. Those ones and zeros may represent a number, a character or a series of flags. An example byte of binary data containing 1100 0001 could be interpreted as:

- Binary data – 1100 0001 or
- Character data 'A' or
- Numeric data '193'

Which of these interpretations a program uses depends upon the instruction that references the storage. For example, Load Halfword (LH) interprets the storage reference as a two-byte signed binary number. The data type you provide with a DS instruction is essentially there to tell a human reader what the area is for. The data type you provide with a DC is essentially there to tell the assembler how to interpret the value.

Simple examples

1 Define an EBCDIC character string initialised to “HAMPSHIRE”

```
DC CE'HAMPSHIRE'
```

2 Define the same, but as ASCII

```
DC CA'HAMPSHIRE'
```

3 Define a packed decimal field initialised to “305”

```
DC PL2'305'
```

4 Define a numeric string initialised to “305”

```
DC Z'305'
```

5 Define a decimal floating point number initialised to 35.92

```
DC EDE3'35.92'
```

Lets examine these declarations in greater detail.

Data Types

HLASM allows the assembler programmer to describe storage areas by its data types. Data types allowed include:

Address – contains the address of a storage area
Binary – contains bit patterns which may be used as flags
Character – contains character strings
Decimal – contains decimal number (also referred to as 'packed decimal')
Fixed-point – contains fixed-point binary number
Floating-point – contains floating point data
Graphic – contains strings that contain pure double-byte data
Hexadecimal – contains large bit patterns entered as hexadecimal
Zoned – contains numeric characters

DC/DS Instructions

The HLASM Programming Reference describes the DC instruction as: '... causes the assembler to generate the binary representation of the data constant you specify into a particular location in the assembled source module' and the DS instruction as 'Reserves areas of storage...'.

A data item may be encoded using several different variations of DC/DS instruction. Using correctly defined variables reduces the likelihood of program errors; reduces the time required to understand the program; reduces the effort required when the program is updated (by others).

The two assembler instructions are similar in format:

symbol **DC** *operand(s)*
symbol **DS** *operand(s)*

Note: DC is almost always preferred over DS. Reasons include:

- Providing a nominal value is useful as "commentary" even when (for example) DC is used in a DSECT.
- When a DC forces alignment that is not already as required, the assembler inserts binary zeros to pad. This makes the load module binary cleaner. It also shows in the assembler listing making inadvertent padding more obvious.

Symbols

symbol provides a label that your program can use to refer to a location within the program. For example, given the following definition:

order_status **DC** *operand(s)*

order_status is the name used internally by the assembler program to address a memory location. The shape and size – the description, is specified with the *operand(s)*.

The *symbol* can consist of 1 to 63 alphanumeric characters, the first of which must be alphabetic (and some special characters such as @). Examples of *symbol* include:

```
DateOfBirth  DC Z'19850913'
TownOfBirth  DC CL32'LONDON'
CountryOfBirth DC CL32'UNITED KINGDOM'
```

Operands

operand(s) define how the storage area is viewed and interpreted.

There are six *operand(s)* which HLASM uses to describe storage:
1. Duplication factor
2. Type
3. Type extension
4. Program type
5. Modifier
6. Nominal value

What do all these operands mean? Do we need to specify all of them? Lets look at the terms listed above.

The **Duplication factor** is a numeric value which causes the storage item to be repeated the number of times as indicated by the number specified. If you do not provide a number, the default value of one is used. If you wanted to allocate a single item of eight character bytes you would code:

```
Variable1  DC CL8'EIGHT'   (implied duplication factor of 1
```

but if you wanted two 8 byte fields, code:

```
Variable1  DC 2CL8'EIGHT'  (explicit duplication factor of 2
```

A duplication factor of zero is allowed. A zero causes the next storage item to be aligned. No storage is allocated other than that required to gain the requested alignment. For example:

```
AligneMe   DC 0D'0'        (storage alignment
Variable1  DC 2CL8'EIGHT'  (variable aligned on boundary
```

Variable *AligneMe* uses a type of 'D' (which we'll describe below), causes the variable *Variable1* to be on a double word boundary. A double word boundary is one which is devisable by eight: 8, 16, 24, 32...

The **Type** is a value that informs the assembler what type of storage is being defined. For example, type 'C' means character data, Each character occupies 1 byte (8 bits). Type 'G' defines graphic data items, each graphic character occupies 2 bytes (16 bits). You must specify the type. For a list of valid types, see Data types below.

The **Type extension** is used by the assembler in conjunction with the type to determine how to interpret the variable and translate it into the correct format. The type extension is optional. For example, you can define two variables, both to be initialised to the character string 'HELLO WORLD', however, one will be in EBCDIC values, the other in ASCII:

```
Variable1  DC CE'HELLO WORLD'
Variable2  DC CA'HELLO WORLD'
```

The assembler generates the following:

```
  Loc  Object Code            Stmt   Source Statement
00004E C8C5D3D3D640E6D6        327 Variable1  DC CE'HELLO WORLD'
000059 48454C4C4F20574F        328 Variable2  DC CA'HELLO WORLD'
```

For a list of type extensions, see type extensions below.

The **Program type** is a user assigned value associated with the variable. Your assembler program can determine the assigned type using the SYSATTRP builtin function. The assembler itself does not use this value. The value is coded as P(*program type*). For example, add program types of 'E1' and 'A1' to the variables:

```
Variable1  DC CEP(C'E1')'HELLO WORLD'
Variable2  DC CAP(C'A1')'HELLO WORLD'
```

The **Modifier** describes the length in bits or bytes of your variable. If the modifier is specified, the variable has an explicit length. If it is omitted, the assembler generates an implied length based on its type and type extension. There are three modifiers: (L) the length modifier, (S) the scale modifier and (E) the exponent modifier.

The length modifier (L) can be used to specify the length, for example:

```
Variable1 DC CEL12'HELLO'
```

will define a variable that has an explicit length of 12 bytes, initialised to the EBCDIC value of `'HELLO'`.

The scale modifier (S) is only used with the fixed-point or floating-point variables. Scaling for fixed point binary is reasonably easy to understand. In simple terms, Sn means the rightmost n bits are used for the fractional part of the value. For example:

```
Scale1 DC FS24'6'
```

puts 6, (binary 00000110) in the high order byte of a fullword (leaving the 24 low order bits for a fractional part).

The exponent modifier (E) is only used with fixed-point or floating-point variables and indicates the power of 10 by which the constant is to be multiplied before conversion to its internal binary format. For example, a binary number with a value of 1,500,000:

```
Expon1 DC FE6'1.5'
```

The assembler generates the following:

```
  Loc  Object Code       Stmt   Source Statement
0000D4 06000000           331 Scale1      DC FS24'6'
0000D8 0016E360           332 Expon2      DC FE6'1.5'
```

The **Nominal value** is the value that is used to initialise the variable. It is the binary representation of the constant. We have already seen numerous examples, such as the EBCDIC value of *'HELLO WORLD'* in:

```
Variable1  DC CE'HELLO WORLD'
```

More Examples

An EBCDIC character string initialised to 'HLASM'. There are several ways of defining this field. Here are just a few:

```
Prod_Name1  DC C'HLASM'            (implied length of 5 bytes
Prod_Name2  DC CL5'HLASM'          (explicit length of 5 bytes
Prod_Name3  DC X'C8D3C1E2D4'       (implied length 5 bytes, hexadecimal
```

A Unicode field initialised to 'HELLO WORLD':

```
Variable3   DC CU'HELLO WORLD'     (implied length 22 bytes, Unicode
```

Several numeric fields initialised to the number 501:

```
Prod_Price1  DC Z'501'        ( implied length 3 bytes
Prod_Price2  DC ZL4'0501'     ( explicit length 4 bytes
Prod_Price3  DC P'501'        ( implied length 2 bytes
Prod_Price4  DC F'501'        ( explicit length 4 bytes
Prod_Price5  DC H'501'        ( explicit length 2 bytes
```

The assembler generates the following:

```
  Loc  Object Code       Stmt   Source Statement
000068 C8D3C1E2D4         317 Prod_name1  DC C'HLASM'
00006D C8D3C1E2D4         318 Prod_name2  DC CL5'HLASM'
000072 C8D3C1E2D4         319 Prod_name3  DC X'C8D3C1E2D4'
000077 F5F0C1             320 Prod_Price1 DC Z'501'
00007A F0F5F0C1           321 Prod_Price2 DC ZL4'0501'
00007E 501C               322 Prod_Price3 DC P'501'
000080 000001F5           323 Prod_Price4 DC F'501'
000084 01F5               324 Prod_Price5 DC H'501'
000086 00480045004C004C   325 Variable3   DC CU'HELLO WORLD'
```

All of the character definitions have resulted in the same object code. However, the numeric definitions differing storage interpretations.

Lets look at those numbers a little closer.

Definition: `000077 F5F0C1 320 Prod_Price1 DC Z'501'`
has a resulted in 3 bytes of storage being allocated. One byte per digit. The storage is not aligned on a word nor halfword boundary. This is zoned data. The top four bits of the last byte hold the sign. The assembler assumes an implied length of 3.

Definition: `00007A F0F5F0C1 321 Prod_Price2 DC ZL4'0501'`
is four bytes in length, and shows the left-pad—with-zeros that applies to zoned format.

Definition: `00007E 501C 322 Prod_Price3 DC P'501'`
occupies 2 bytes. Each 4 bits holds a numeric value. The final 4 bits contain the sign.

Definition: `000080 000001F5 323 Prod_Price4 DC F'501'`
occupies 4 bytes (a word). The number is stored in hexadecimal format. The storage is aligned on a word boundary.

Definition: `000084 01F5 324 Prod_Price5 DC H'501'`
is the same as the one above except that it occupies 2 bytes (a half word) and is aligned on a half word boundary.

Definition: `000086 00480045004C004C 325 Variable3 DC CU'HELLO WORLD'`
occupies 22 bytes, is in Unicode format.

Manipulating Variables

Once our storage has been declared using DC/DS instructions, you use machine instructions to manipulate the variables.

Adding simple numbers

Some storage formats require different flavours of machine instructions to manipulate them. For example, let us add 5 to to the following variables:

```
Prod_Price3  DC P'501'    Packed
Prod_Price4  DC F'501'    Fullword
Prod_Price5  DC H'501'    Halfword
```

1 Packed Decimal storage

```
AP   Prod_Price3,=p'5'      Add 5 using the Decimal instructions
```

2 Fixed point – fullword

```
L    R1,Prod_Price4         Load into register
AHI  R1,5                   Add 5
ST   R1,Prod_Price4         Save result
```

3 Fixed point – halfword

```
LH   R1,Prod_Price5          Load into register
AHI  R1,5                    Add 5
STH  R1,Prod_Price5          Save result
```

For both 2) and 3) above, several different sequences of instructions could have been used.

As you can see, depending upon the data type, a different instruction is used.

Manipulating the same data type is relatively easy. But what about add differing types? Lets add the value in variable *Order_Count1* to *Order_Count2* and store the value in *Order_Total*.

Order_Count1 is a packed decimal field, 3 bytes in length, and initialised to 302. *Order_Count2* is a halfword, initialised to 301. *Order_Total* is a fullword. Because the variables are of differing types, they will need to be converted.

```
Order_Count1 DC PL3'302'          packed decimal
Order_Count2 DC H'301'            halfword
Order_Total  DS F                 fullword
```

Step 1: Convert *Order_Count1* from packed decimal to fixed point binary.

```
ZAP   WORK1,Order_Count1 Copy to 8 byte work field
CVB   R1,WORK1           Convert packed to fixed point bin
```

Because the CVB (convert to binary) instruction requires an 8 byte input field, we have to copy the 3 byte *Order_Count1* to a work field *WORK1* declared as:

```
WORK1        DS PL8               packed decimal work field
```

The converted data is saved in register 1. Its worth noting that some instructions, like CVB, simply use a fixed number of bytes at the specified location but others, like ZAP, get the number of bytes from the DC or DS declaration.

Step 2: Add the halfword *Order_Count2* to register 1.

```
 AH    R1,Order_Count2    Add the halfword
```

Step 3: Store the result which is in register 1 to the fullword *Order_Total*

```
ST    R1,Order_Total     and save result
```

The full program can be found below.

The Work Field

`WORK1` was defined as a 8 byte packed decimal field because it is a requirement of the CVB instruction. However, other 8 byte declarations such as:

```
WORK1 DS D
```

could have been used.

Either of these declarations would have worked, but if this program was written by a colleague and you had to make a change to the source code, which declaration would you prefer ?

Data types

Code	Constant type	Machine format
C	Character	8-bit code for each character
G	Graphic	16-bit code for each character
X	Hexadecimal	4-bit code for each hexadecimal digit
B	Binary	Binary format
F	Fixed-point	Signed, fixed-point binary format; normally a fullword
H	Fixed-point	Signed, fixed-point binary format; normally a halfword
E	Floating-point	Short floating-point format; normally a fullword
D	Floating-point	Long floating-point format; normally a doubleword
L	Floating-point	Extended floating-point format; normally two doublewords
P	Decimal	Packed decimal format
Z	Decimal	Zoned decimal format
A	Address	Value of address; normally a fullword
Y	Address	Value of address; normally a halfword
S	Address	Base register and displacement value; a halfword
V	Address	Space reserved for external symbol addresses; normally a fullword
J	Address	Space reserved for length of class or DXD; normally a fullword
Q	Address	Space reserved for external dummy section offset
R	Address	Space reserved for PSECT addresses; normally a fullword

Type Extensions

Type	Type extension	
C	A	ASCII character constant
	E	EBCDIC character constant
	U	Unicode UTF-16 character constant
E	H	Hexadecimal floating-point constant
	B	Binary floating-point constant
	D	Decimal floating-point constant
D	H	Hexadecimal floating-point constant
	B	Binary floating-point constant
	D	Decimal floating-point constant
L	H	Hexadecimal floating-point constant
	B	Binary floating-point constant
	D	Decimal floating-point constant
	Q	Hexadecimal floating-point, quadword alignment
F	D	Doubleword fixed-point constant
A	D	Doubleword address constant
V	D	Doubleword address constant
J	D	Doubleword address constant
Q	D	Doubleword address constant
Q	Y	20-bit address constant
R	D	Doubleword address constant

Sample Program

```
NUM2      CSECT
          SYSSTATE ARCHLVL=2
          COPY  IEABRCX
R1        EQU   1
R12       EQU   12
R14       EQU   14
R15       EQU   15
          SAVE (14,12),,*
          LARL  R12,STORAGEAREA
          USING STORAGEAREA,R12
          ZAP   WORK1,Order_Count1 Copy to 8 byte work field
          CVB   R1,WORK1           Convert packed to fixed point bin
          AH    R1,Order_Count2    Add the half word
          ST    R1,Order_Total     and save result
          RETURN (14,12),,RC=0     return
STORAGEAREA  DC 0D'0'              align storage
Order_Count1 DC PL3'302'           packed decimal
Order_Count2 DC H'301'             halfword
Order_Total  DS F                  fullword
WORK1        DS PL8                packed decimal work field
          END   NUM2
```