How to Decode PMAC's Gather Buffer

This application note gives details on decoding PMAC's gather buffer. A prerequisite to decoding the information that PMAC has stored in its gather buffer is to first understand how PMAC reports this data to the host. Once this is done, the pseudo-code along with some examples at the end of this document will be better understood.

PMAC reports the gathered data starting with the first recorded sample followed by the second and so on. A sample may consist from 1 to 24 sources as specified by a PMAC I-variable, I20. The location of the source and which word of PMAC's dual memory (X, Y, or both) is to be gathered is specified by I21 - I44. If a source specification mandates that a word is to be gathered (i.e. either the X or Y memory space, 24 bits, six hex characters) then only a long (four byte) integer value may be decoded from that particular source. If a double word is specified on the other hand (i.e. both X and Y, 48 bits, 12 hex digits) for a particular source, then it may be decoded as either a double floating point value or a long integer. (The PMAC memory map will specify either L for double or D for long). The order in which the sources are reported within a given sample is the same as the order specified by the I-Variables I21 - 24.

For example, if both I21 and I22 are to be gathered, first I21's source will be reported and then I22's, then the next sample will be reported starting with I21's source, etc. PMAC reports the data in 12 hexadecimal (48-bits) digit chunks at a time. A typical example of how PMAC will report the samples of two sources, both single words, is shown below:

First Sample		Second Sample		Third Sample	
First source	Second source	First source	Second source	First source	Second source
ABC123	FFF000	ABC124	FFF001	ABC124	
FFF001	etc.				

If an odd number of words are specified to be gathered, PMAC will always make the sample size even by appending the end of it with the servo cycle timer, a 24-bit memory location. For example, PMAC was told to retrieve three sources where the first source is a double word, the second is a single word and the last one is a double word. PMAC will report the information as shown below:

First Sample		First Sample		First Sample	
First source	First source	Second source	Third source	Third source	Servo Cycle Counter
000001	FEDCBA	ABC124	FFF001	ABC124	
000123					
Second Sample		**Second Sample**		**Second Sample**	
First source	First source	Second source	Third source	Third source	Servo Cycle Counter
000002	FEDCBB	ABC125	FFF002	ABC125	000245
etc.					

Decoding Individual Samples

Usually the data is brought into a character array and the information is processed from there. The `Hex_to_Decimal()` function is utilized in the decoding algorithms is shown in C at the end of this document.

Decoding a Single Word

The algorithm is expressed in pseudo-code. It is assumed that the programmer has placed the 6-digit hexadecimal number inside the character array buffer (i.e. buffer = 280000). The variable value_low will contain the converted data.

```
/* Variable declarations: */
long value_low; /* A 4 byte integer */
value_low = Hex_to_Decimal(buffer,6); if(value_low >= 838608)
value_low = value_low - 16777216;
```

Example:
The string 280000 is to be decoded. The following calculations would be performed to get the decoded data:

```
value_low = Hex_to_Decimal("280000",6")
result: value low = 2621440
```

Since value_low is not greater than or equal to 838608 no further calculations need to be done and the result is 2621440.

Decoding a Double Word

Again, there are two possible ways to decode the 48-bit, 12-digit double word. The following explains how to convert data of this type to a meaningful epic of information.

value_low

Mantissa Exponent

```
/* Variable declarations: */
long value_hi, value_low, low_value, value; integer exponent;
     value_hi = Hex_to_Decimal(MANTISA,8); value_low =
     Hex_to_Decimal(LOW_VALUE,1); exponent = Hex_to_Decimal(EXP) - 2047;
     value = ( (16 * value_hi) +
value_low))/34359738368.0
     value = value * (1 << exponent);
```

The last line is an epic of C code. (1 << exponent) is equivalent to shifting a register which has a value of 1 in it exponent number of places to the left. For example: (1<< 2) = 4;

Example:
Now decode the string 4000000000805.

value_low

Mantissa Exponent

```
value_hi = Hex_to_Decimal("400000000",8)
result: value_hi = 1073741824
value_low = Hex_to_Decimal("0",1);
result: value_low = 0
exponent = Hex_to_Decimal ("805",3) - 2047
result: exponent = 6
value = (16 * 1073741824 + 0 ) / 34359738368
result: value = .5
value = .5 * (1<<6) = .5 * 64
result: value = 32
*/
value = -1248
```

Hence `4000000000805` is equivalent to 32.0.

The next piece of pseudo code details how to decode a 12-hexadecimal digit string to a long integer. The string which holds the unconverted data is shown below: Low_word

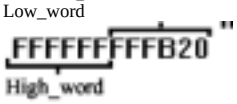
High_word

```
/* Variable declarations:
long value_low, value_high, value; /* A 4 byte initeger */
value_low = Hex_to_Decimal[Low_word] value_high = Hex_to_Decimal[_High_word]
if(value_high > 8388608)
```

```
value_high = value_high - 16777216 value = 16777216.0 * value_high +
value_low
```

Example:

Low_word



Decode the 12-digit string FFFFFFFFFB20 to a long integer:

```
value_low = Hex_to_Decimal("FFFB20",6); result: value_low = 16775968
value_high = Hex_to_Decimal("FFFFFF",8) result: value_high = 16777215
```

Since `value_high` is greater than 8388608, subtract 16777216 from the result.

```
value_high = 8388608 - 16777216 result: value_high = -1
value = 16777216 * -1 + 167775968
```

Hex_to_Decimal

```
long Hex_to_Decimal(char *in_str, int str_ln) //Converts an input (in_str) in hex
{                                             //to a long (out_num)
long out_num = 0;
int i;
for(i = 0; i < str_ln; i++){ //Convert each hex char to a dec num and add
      if(in_str[i] >= 'A')                    //For an alpha char A - F
            out_num = (out _num << 4)     + (in_str[i] - 55);
      else                                    //For a num 0 – 9
            out_num = (out_num << 4)      + (in_str[i] - 48);
      return(out_num);
```