Structures in Assembly Language Programs

Structures, or records, are an abstract data type that allows a programmer to collect different objects together into a single, composite, object. Structures can help make programs easier to read, write, modify, and maintain. Used appropriately, they can also help your programs run faster. Despite the advantages that structures offer, their appearance in assembly language is a relatively recent phenomenon (in the past two decades, or so), and many assemblers still do not support this facility. Furthermore, many "old-timer" assembly language programmers attempt to argue that the appearance of records violates the whole principle of "assembly language programming." This article will certain refute such arguments and describe the benefits of using structures in an assembly language program.

Despite the fact that records have been available in various assembly languages for years (e.g., Microsoft's MASM assembler introduced structures in 80x86 assembly language in the 1980s), the "lack of support for structures" is a common argument against assembly language by HLL programmers who don't know much about assembly. In some respects, their ignorance is justified -- many assemblers don't support structures or records. A second goal of this article is to educate assembly language programmers to counter claims like "assembly language doesn't support structures." Hopefully, that same education will convince those assembly language programmers who've never bothered to use structures, to consider their use.

This article will use the term "record" to denote a structure/record to avoid confusion with the more general term "data structure". Note, however, that the terms "record" and "structure" are synonymous in this article.

What is a Record (Structure)?

The whole purpose of a record is to let you encapsulate different, but logically related, data into a single package. Here is a typical record declaration, in HLA using the RECORD / ENDRECORD declaration:

```
type
   student:
            record
               Name:       string;
               Major:      int16;
               SSN:        char[12];
               Midterm1:   int16;
               Midterm2:   int16;
               Final:      int16;
               Homework:   int16;
               Projects:   int16;
            endrecord;
```

The field names within the record must be unique. That is, the same name may not appear two or more times in the same record. However, in reasonable assemblers (like HLA) that support true structures, all the field names are local to that record. With such assemblers, you may reuse those field names elsewhere in the program.

The RECORD/ENDRECORD type declaration may appear in a variable declaration section (e.g., an HLA STATIC or VAR section) or in a TYPE declaration section. In the previous example the *Student* declaration appears in an HLA TYPE section, so this does not actually allocate

any storage for a *Student* variable. Instead, you have to explicitly declare a variable of type *Student*. The following example demonstrates how to do this:

```
var
   John: Student;
```

This allocates 28 bytes of storage: four bytes for the Name field (HLA strings are four-byte pointers to character data found elsewhere in memory), 12 bytes for the SSN field, and two bytes for each of the other six fields.

If the label *John* corresponds to the *base address* of this record, then the *Name* field is at offset *John+0*, the *Major* field is at offset *John+4*, the *SSN* field is at offset *John+6*, etc.

To access an element of a structure you need to know the offset from the beginning of the structure to the desired field. For example, the *Major* field in the variable *John* is at offset 4 from the base address of *John*. Therefore, you could store the value in AX into this field using the instruction

```
mov( ax, (type word John[4]) );
```

Unfortunately, memorizing all the offsets to fields in a record defeats the whole purpose of using them in the first place. After all, if you've got to deal with these numeric offsets why not just use an array of bytes instead of a record?

Well, as it turns out, assemblers like HLA that support true records commonly let you refer to field names in a record using the same mechanism C/C++ and Pascal use: the dot operator. To store AX into the *Major* field, you could use "mov(ax, John.Major);" instead of the previous instruction. This is much more readable and certainly easier to use.

Record Constants

HLA lets you define record constants. In fact, HLA is probably unique among x86 assemblers insofar as it supports both symbolic record constants and literal record constants. Record constants are useful as initializers for static record variables. They are also quite useful as compile-time data structures when using the HLA compile-time language (that is, the macro processor language). This section discusses how to create record constants.

A record literal constant takes the following form:

RecordTypeName:[*List_of_comma_separated_constants*]

The *RecordTypeName* is the name of a record data type you've defined in an HLA TYPE section prior to this point. To create a record constant you must have previously defined the record type in a TYPE section of your program.

The constant list appearing between the brackets are the data items for each of the fields in the specified record. The first item in the list corresponds to the first field of the record, the second item in the list corresponds to the second field, etc. The data types of each of the constants appearing in this list must match their respective field types. The following example demonstrates how to use a literal record constant to initialize a record variable:

```
type
   point:
         record
            x:int32;
            y:int32;
```

```
            z:int32;
         endrecord;

static
   Vector: point := point:[ 1, -2, 3 ];
```

This declaration initializes *Vector.x* with 1, *Vector.y* with -2, and *Vector.z* with 3.

You can also create symbolic record constants by declaring record objects in the CONST or VAL sections of an HLA program. You access fields of these symbolic record constants just as you would access the field of a record variable, using the dot operator. Since the object is a constant, you can specify the field of a record constant anywhere a constant of that field's type is legal. You can also employ symbolic record constants as record variable initializers. The following example demonstrates this:

```
type
   point:
         record
            x:int32;
            y:int32;
            z:int32;
         endrecord;

const
   PointInSpace: point := point:[ 1, 2, 3 ];

static
   Vector: point := PointInSpace;
   XCoord: int32 := PointInSpace.x;
```

Arrays of Records

It is a perfectly reasonable operation to create an array of records. To do so, you simply create a record type and then use the standard array declaration syntax when declaring an array of that record type. The following example demonstrates how you could do this:

```
type
   recElement:
      record
         << fields for this record >>
      endrecord;
      .
      .
      .
static
   recArray: recElement[4];
```

Naturally, you can create multidimensional arrays of records as well. You would use the standard row or column major order functions to compute the address of an element within such records. The only thing that really changes (from the discussion of arrays) is that the size of each element is the size of the record object.

```
static
   rec2D: recElement[ 4, 6 ];
```

Arrays and Records as Record Fields

Records may contain other records or arrays as fields. Consider the following definition:

```
type
   Pixel:
      record
         Pt:         point;
         color:      dword;
      endrecord;
```

The definition above defines a single point with a 32 bit color component. When initializing an object of type Pixel, the first initializer corresponds to the *Pt* field, *not the x-coordinate field*. **The following definition is incorrect:**

```
static
    ThisPt: Pixel := Pixel:[ 5, 10 ];   // Syntactically incorrect!
```

The value of the first field (“5”) is not an object of type *point*. Therefore, the assembler generates an error when encountering this statement. HLA will allow you to initialize the fields of *Pixel* using declarations like the following:

```
static
    ThisPt: Pixel := Pixel:[ point:[ 1, 2, 3 ], 10 ];
    ThatPt: Pixel := Pixel:[ point:[ 0, 0, 0 ], 5 ];
```

Accessing *Pixel* fields is very easy. Like a high level language you use a single period to reference the *Pt* field and a second period to access the *x*, *y*, and *z* fields of *point*:

```
      stdout.put( “ThisPt.Pt.x = “, ThisPt.Pt.x, nl );
      stdout.put( “ThisPt.Pt.y = “, ThisPt.Pt.y, nl );
      stdout.put( “ThisPt.Pt.z = “, ThisPt.Pt.z, nl );
       .
       .
       .
   mov( eax, ThisPt.Color );
```

You can also declare *arrays* as record fields. The following record creates a data type capable of representing an object with eight points (e.g., a cube):

```
type
   Object8:
      record
         Pts:        point[8];
         Color:      dword;
      endrecord;
```

There are two common ways to nest record definitions. As noted earlier in this section, you can create a record type in a TYPE section and then use that type name as the data type of some field within a record (e.g., the *Pt:point* field in the *Pixel* data type above). It is also possible to

declare a record directly within another record without creating a separate data type for that record; the following example demonstrates this:

```
type
   NestedRecs:
      record
         iField: int32;
         sField: string;
         rField:
            record
               i:int32;
               u:uns32;
            endrecord;
         cField:char;
      endrecord;
```

Generally, it's a better idea to create a separate type rather than embed records directly in other records, but nesting them is perfectly legal and a reasonable thing to do on occasion.

Controlling Field Offsets Within a Record

By default, whenever you create a record, most assemblers automatically assign the offset zero to the first field of that record. This corresponds to records in a high level language and is the intuitive default condition. In some instances, however, you may want to assign a different starting offset to the first field of the record. The HLA assembler provides a mechanism that lets you set the starting offset of the first field in the record.

The syntax to set the first offset is

```
name:
    record := startingOffset;
        << Record Field Declarations >>
    endrecord;
```

Using the syntax above, the first field will have the starting offset specified by the *startingOffset int32* constant expression. Since this is an *int32* value, the starting offset value can be positive, zero, or negative.

One circumstance where this feature is invaluable is when you have a record whose base address is actually somewhere within the data structure. The classic example is an HLA string. An HLA string uses a record declaration similar to the following:

```
   record
      MaxStrLen: dword;
      length: dword;
      charData: char[xxxx];
   endrecord;
```

However, HLA string pointers do not contain the address of the *MaxStrLen* field; they point at the *charData* field. The str.strRec record type found in the HLA Standard Library Strings module uses a record declaration similar to the following:

```
type
```

```
   strRec:
      record := -8;
         MaxStrLen: dword;
         length:    dword;
         charData:  char;
      endrecord;
```

The starting offset for the *MaxStrLen* field is -8. Therefore, the offset for the *length* field is -4 (four bytes later) and the offset for the *charData* field is zero. Therefore, if EBX points at some string data, then "(type str.strRec [ebx]).length" is equivalent to "[ebx-4]" since the *length* field has an offset of -4.

Aligning Fields Within a Record

To achieve maximum performance in your programs, or to ensure that your records properly map to records or structures in some high level language, you will often need to be able to control the alignment of fields within a record. For example, you might want to ensure that a *dword* field's offset is an even multiple of four. You use the ALIGN directive in a record declaration to do this. The following example shows how to align some fields on important boundaries:

```
type
   PaddedRecord:
      record
         c: char;
            align(4);
         d: dword;
         b: boolean;
            align(2);
         w: word;
      endrecord;
```

Whenever HLA encounters the ALIGN directive within a record declaration, it automatically adjusts the following field's offset so that it is an even multiple of the value the ALIGN directive specifies. It accomplishes this by increasing the offset of that field, if necessary. In the example above, the fields would have the following offsets: *c*:0, *d*:4, *b*:8, *w*:10.

If you want to ensure that the record's size is a multiple of some value, then simply stick an ALIGN directive as the last item in the record declaration. HLA will emit an appropriate number of bytes of padding at the end of the record to fill it in to the appropriate size. The following example demonstrates how to ensure that the record's size is a multiple of four bytes:

```
type
   PaddedRec:
      record
         << some field declarations >>

         align(4);

      endrecord;
```

Be aware of the fact that the ALIGN directive in a RECORD only aligns fields in memory if the record object itself is aligned on an appropriate boundary. Therefore, you must ensure appropriate alignment of any record variable whose fields you're assuming are aligned.

If you want to ensure that all fields are appropriately aligned on some boundary within a record, but you don't want to have to manually insert ALIGN directives throughout the record, HLA provides a second alignment option to solve your problem. Consider the following syntax:

```
type
  alignedRecord3 :
    record[4]
      << Set of fields >>
    endrecord;
```

The "[4]" immediately following the RECORD reserved word tells HLA to start all fields in the record at offsets that are multiples of four, regardless of the object's size (and the size of the objects preceeding the field). HLA allows any integer expression that produces a value in the range 1..4096 inside these parenthesis. If you specify the value one (which is the default), then all fields are packed (aligned on a byte boundary). For values greater than one, HLA will align each field of the record on the specified boundary. For arrays, HLA will align the field on a boundary that is a multiple of the array element's size. The maximum boundary HLA will round any field to is a multiple of 4096 bytes.

Note that if you set the record alignment using this syntactical form, any ALIGN directive you supply in the record may not produce the desired results. When HLA sees an ALIGN directive in a record that is using field alignment, HLA will first align the current offset to the value specified by ALIGN and then align the next field's offset to the global record align value.

Nested record declarations may specify a different alignment value than the enclosing record, e.g.,

```
type
   alignedRecord4 : record[4]
      a:byte;
      b:byte;
      c:record[8]
         d:byte;
         e:byte;
      endrecord;
      f:byte;
      g:byte;
   endrecord;
```

In this example, HLA aligns fields a, b, f, and g on dword boundaries, it aligns d and e (within c) on eight-byte boundaries. Note that the alignment of the fields in the nested record is true only within that nested record. That is, if c turns out to be aligned on some boundary other than an eight-byte boundary, then d and e will not actually be on eight-byte boundaries; they will, however be on eight-byte boundaries relative to the start of c.

In addition to letting you specify a fixed alignment value, HLA also lets you specify a minimum and maximum alignment value for a record. The syntax for this is the following:

```
type
   recordname : record[maximum : minimum]
      << fields >>
```

```
   endrecord;
```

Whenever you specify a maximum and minimum value as above, HLA will align all fields on a boundary that is at least the minimum alignment value. However, if the object's size is greater than the minimum value but less than or equal to the maximum value, then HLA will align that particular field on a boundary that is a multiple of the object's size. If the object's size is greater than the maximum size, then HLA will align the object on a boundary that is a multiple of the maximum size. As an example, consider the following record:

```
type
   r: record[ 4:1 ];
      a:byte;           // offset 0
      b:word;           // offset 2
      c:byte;           // offset 4
      d:dword[2];       // offset 8
      e:byte;           // offset 16
      f:byte;           // offset 17
      g:qword;          // offset 20
   endrecord;
```

Note that HLA aligns `g` on a dword boundary (not qword, which would be offset 24) since the maximum alignment size is four. Note that since the minimum size is one, HLA allows the `f` field to be aligned on an odd boundary (since it's a byte).

If an array, record, or union field appears within a record, then HLA uses the size of an array element or the largest field of the record or union to determine the alignment size. That is, HLA will align the field without the outermost record on a boundary that is compatible with the size of the largest element of the nested array, union, or record.

HLA sophisticated record alignment facilities let you specify record field alignments that match that used by most major high level language compilers. This lets you easily access data types used in those HLLs without resorting to inserting lots of ALIGN directives inside the record.

Using Records/Structures in an Assembly Language Program

In the "good old days" assembly language programmers typically ignored records. Records and structures were treated as unwanted stepchildren from high-level languages, that weren't necessary in "real" assembly language programs. Manually counting offsets and hand-coding literal constant offsets from a base address was the way "real" programmers wrote code in early PC applications. Unfortunately for those "real programmers", the advent of sophisticated operating systems like Windows and Linux put an end to that nonsense. Today, it is very difficult to avoid using records in modern applications because too many API functions require their use. If you look at typical Windows and Linux include files for C or assembly language, you'll find hundreds of different structure declarations, many of which have dozens of different members. Attempting to keep track of all the field offsets in all of these structures is out of the question. Worse, between various releases of an operating system (e.g., Linux), some structures have been known to change, thus exacerbating the problem. Today, it's unreasonable to expect an assembly language programmer to manually track such offsets - most programmers have the reasonable expectation that the assembler will provide this facility for them.

Implementing Structures in an Assembler

Unfortunately, properly implementing structures in an assembler takes considerable effort. A large number of the "hobby" (i.e., non-commercial) assemblers were not designed from the start to support sophisticated features such as records/structures. The symbol table management routines in most assemblers use a "flat" layout, with all of the symbols appearing at the same level in the symbol table database. To properly support structures or records, you need a hierarchical structure in your symbol table database. The bad news is that it's quite difficult to retrofit a hierarchical structure over the top of a flat database (i.e., the symbol "hobby assembler" symbol table). Therefore, unless the assembler was originally designed to handle structures properly, the result is usually a major hacked-up kludge.

Four assemblers I'm aware of, MASM, TASM, OPTASM, and HLA, handle structures well. Most other assemblers are still trying to simulate structures using a flat symbol table database, with varying results.

Probably the first attempt people make at records, when their assembler doesn't support them properly, is to create a list of constant symbols that specify the offsets into the record. Returning to our first example (in HLA):

```
type
   student:
            record
               Name:       string;
               Major:      int16;
               SSN:        char[12];
               Midterm1:   int16;
               Midterm2:   int16;
               Final:      int16;
               Homework:   int16;
               Projects:   int16;
            endrecord;
```

One attempt might be the following:

```
const
    Name := 0;
    Major := 4;
    SSN := 6;
    Midterm1 := 18;
    Midterm2 := 20;
    Final := 22;
    Homework := 24;
    Projects := 26;
    size_student := 28;
```

With such a set of declarations, you could reserve space for a student "record" by reserving "size_student" bytes of storage (which almost all assemblers handle okay) and then you can access fields of the record by adding the constant offset to your base address, e.g.,

```
static
    John : byte[ size_student ];
      .
      .
      .
```

```
mov( John[Midterm1], ax );
```

There are several problems with this approach. First of all, the field names are global and must be globally unique. That is, you cannot have two record types that have the same fieldname (as is possible with the assembler supports true records). The second problem, which is fundamentally more problematic, is the fact that you can attach these constant offsets to any object, not just a "student record" type object. For example, suppose "ClassAverage" is an array of words, there is nothing stopping you from writing the following when using constant equate values to simulate record offsets:

```
mov( ClassAverage[ Midterm1 ], ax );
```

Finally, and probably the most damning criticism of this approach, is that it is very difficult to maintain code that accesses structures in this manner. Inserting fields into the middle of a record, changing data types, and coming up with globally unique names can create all sorts of problems. Many high-level language programmers who've tried to learn assembly language have given up after discovering that they had to maintain records in this fashion in an assembly language program (too bad they didn't start off with a reasonable assembler that properly supports structures).

Manually maintaining all the constant offsets is a maintenance nightmare. So somewhere along the way, some assembly language programmers figured out that they could write macros to handle the declaration of constant offsets for them. For example, here's how you could do this in an HLA program:

```
program t;


#macro struct( _structName_, _dcls_[] ):
    _dcl_, _id_, _type_, _colon_, _offset_;

    ?_offset_ := 0;
    ?_dcl_:string;
    #for( _dcl_ in _dcls_ )

        ?_colon_ := @index( _dcl_ , 0, ":" );
        #if( _colon_ = -1 )

            #error
            (
                "Expected <id>:<type> in struct definition, encountered: ",
                _dcl_
            )

        #else

            ?_id_ := @substr( _dcl_, 0, _colon_ );
           ?_type_ := @substr( _dcl_, _colon_+1, @length( _dcl_ ) - _colon_ );
            ?@text( _id_ ) := _offset_;
            ?_offset_ := _offset_ + @size( @text( _type_ ));

        #endif;

    #endfor
```

```
    ?_structName_:text := "byte[" + @string( _offset_ ) + "]";

#endmacro

struct( threeItems, i:byte, j:word, k:dword )

static
    aStruct: threeItems;

begin t;

    mov( (type byte aStruct[i]), al );
    mov( (type word aStruct[j]), ax );
    mov( (type dword aStruct[k]), eax );


end t;
```

The "struct" macro expects a set of valid HLA variable declarations supplied as macro arguments. It generates a set of constants using the supplied variable names whose offsets are adjusted according to the size of the objects previously appearing in the list. In this example, HLA creates the following equates:

```
  i = 0
  j = 1
  k = 3
```

This declaration also creates a "data type" named "threeItems" which is equivalent to "byte[7]" (since there are seven bytes in this record) that you may use to create variables of type "threeItems", as is done in this example.

Creating structures with macros solves one of the three major problems: it makes it easier to maintain the constant equates list, as you do not have to manually adjust all the constants when inserting and removing fields in a record. This does not, however, solve the other problems (particularly, the global identifier problem).

While fancier macros could be written, macros that generate identifiers like "objectname_fieldName" that help solve the globally unique problem, the bottom line is that these hacks begin to fail when you attempt to declare nested records, arrays within records, and arrays of records (possibly containing nested records and arrays of records). The bottom line is this: assemblers that don’t properly support structures are going to have problems when you’ve got to work with data structures from high-level languages (e.g., OS API calls, where the OS is written in C, such as Windows and Linux). You’re much better off using an assembler that fully supports structures (and other advanced data types) if you need to use structures in your programs.