Records, Unions, and Name Spaces Chapter Five

5.1 Chapter Overview

This chapter discusses how to declare and use record (structures), unions, and name spaces in your programs. After strings and arrays, records are among the most commonly used composite data types; indeed, records are the mechanism you use to create user-defined composite data types. Many assembly language programmers never bother to learn how to use records in assembly language, yet would never consider not using them in high level language programs. This is somewhat inconsistent since records (structures) are just as useful in assembly language programs as in high level language programs. Given that you use records in assembly language (and especially HLA) in a manner quite similar to high level languages, there really is no reason for excluding this important tool from your programmer's tool chest. Although you'll use unions and name spaces far less often than records, their presence in the HLA language is crucial for many advanced applications. This brief chapter provides all the information you need to successfully use records, unions, and name spaces within your HLA programs.

5.2 Records

Another major composite data structure is the Pascal *record* or C/C++ *structure*[1]. The Pascal terminology is probably better, since it tends to avoid confusion with the more general term *data structure*. Since HLA uses the term "record" we'll adopt that term here.

Whereas an array is homogeneous, whose elements are all the same, the elements in a record can be of any type. Arrays let you select a particular element via an integer index. With records, you must select an element (known as a *field*) by name.

The whole purpose of a record is to let you encapsulate different, but logically related, data into a single package. The Pascal record declaration for a student is probably the most typical example:

```
student =
    record
        Name: string [64];
        Major: integer;
        SSN:   string[11];
        Midterm1: integer;
        Midterm2: integer;
        Final: integer;
        Homework: integer;
        Projects: integer;
    end;
```

Most Pascal compilers allocate each field in a record to contiguous memory locations. This means that Pascal will reserve the first 65 bytes for the name[2], the next two bytes hold the major code, the next 12 the Social Security Number, etc.

In HLA, you can also create structure types using the RECORD/ENDRECORD declaration. You would encode the above record in HLA as follows:

```
type
    student:   record
                   Name: char[65];
                   Major: int16;
                   SSN:   char[12];
```

1. It also goes by some other names in other languages, but most people recognize at least one of these names.
2. Strings require an extra byte, in addition to all the characters in the string, to encode the length.


```
        Midterm1: int16;
        Midterm2: int16;
        Final: int16;
        Homework: int16;
        Projects: int16;
    endrecord;
```

As you can see, the HLA declaration is very similar to the Pascal declaration. Note that, to be true to the Pascal declaration, this example uses character arrays rather than strings for the *Name* and *SSN* (U.S Social Security Number) fields. In a real HLA record declaration you'd probably use a string type for at least the name (keeping in mind that a string variable is only a four byte pointer).

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, all field names are local to that record. Therefore, you may reuse those field names elsewhere in the program.

The RECORD/ENDRECORD type declaration may appear in a variable declaration section (e.g., STATIC or VAR) or in a TYPE declaration section. In the previous example the *Student* declaration appears in the 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 81 bytes of storage laid out in memory as shown in Figure 5.1.

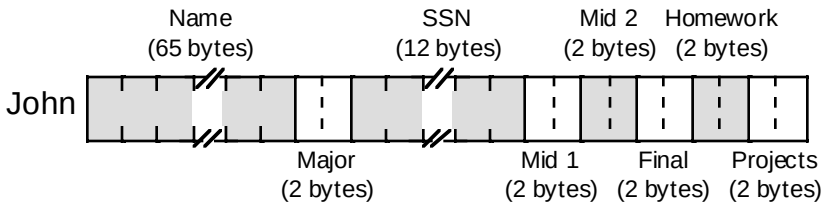

Figure 5.1 Student Data Structure Storage in Memory

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+65*, the *SSN* field is at offset *John+67*, 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 65 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[65]) );
```

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, HLA lets 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.

Note that the use of the dot operator does *not* introduce a new addressing mode. The instruction "mov(ax, John.Major);" still uses the displacement only addressing mode. HLA simply adds the base address of *John* with the offset to the *Major* field (65) to get the actual displacement to encode into the instruction.

Like any type declaration, HLA requires all record type declarations to appear in the program before you use them. However, you don't have to define all records in the TYPE section to create record variables. You can use the RECORD/ENDRECORD declaration directly in a variable declaration section. This is con-

venient if you have only one instance of a given record object in your program. The following example demonstrates this:

```
storage

    OriginPoint:  record
                      x: uns8;
                      y: uns8;
                      z: uns8;
                  endrecord;
```

5.3 Record Constants

HLA lets you define record constants. In fact, HLA 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 (see the chapters on Macros and the HLA Compile-Time 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 your 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;
        .
        .
        .
    stdout.put( "Y Coordinate is ", PointInSpace.y, nl );
        .
        .
        .
```

5.4 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];
```

To access an element of this array you use the standard array indexing techniques found in the chapter on arrays. Since *recArray* is a single dimension array, you'd compute the address of an element of this array using the formula "baseAddress + index*@size(recElement)." For example, to access an element of *recArray* you'd use code like the following:

```
// Access element i of recArray:

    intmul( @size( recElement ), i, ebx );  // ebx := i*@size( recElement )
    mov( recArray.someField[ebx], eax );
```

Note that the index specification follows the entire variable name; remember, this is assembly not a high level language (in a high level language you'd probably use "recArray[i].someField").

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 ];
        .
        .
        .
    // Access element [i,j] of rec2D and load "someField" into EAX:

    intmul( 6, i, ebx );
    add( j, ebx );
    intmul( @size( recElement ), ebx );
    mov( rec2D.someField[ ebx ], eax );
```


5.5 Arrays/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;
```

This record allocates storage for eight different points. Accessing an element of the *Pts* array requires that you know the size of an object of type *point* (remember, you must multiply the index into the array by the size of one element, 12 in this particular case). Suppose, for example, that you have a variable *CUBE* of type *Object8*. You could access elements of the *Pts* array as follows:

```
// CUBE.Pts[i].x := 0;

        mov( i, ebx );
        intmul( 12, ebx );
        mov( 0, CUBE.Pts.x[ebx] );
```

The one unfortunate aspect of all this is that you must know the size of each element of the *Pts* array. Fortunately, HLA provides a built-in function that will compute the size of an array element (in bytes) for you: the *@size* function. You can rewrite the code above using *@size* as follows:

```
// CUBE.Pts[i].x := 0;

        mov( i, ebx );
```



```
        intmul( @size( point ), ebx );
        mov( 0, CUBE.Pts.x[ebx] );
```

This solution is much better than multiplying by the literal constant 12. Not only does HLA figure out the size for you (so you don't have to), it automatically substitutes the correct size if you ever change the definition of the *point* record in your program. For this reason, you should always use the @size function to compute the size of array element objects in your programs.

Note in this example that the index specification ("[ebx]") follows the whole object name even though the array is *Pts*, not *x*. Remember, the "[ebx]" specification is an indexed addressing mode, not an array index. Indexes always follow the entire name, you do not attach them to the array component as you would in a high level language like C/C++ or Pascal. This produces the correct result because addition is commutative, and the dot operator (as well as the index operator) corresponds to addition. In particular, the expression "CUBE.Pts.x[ebx]" tells HLA to compute the sum of *CUBE* (the base address of the object) plus the offset to the *Pts* field, plus the offset to the *x* field plus the value of EBX. Technically, we're really computing offset(*CUBE*)+offset(*Pts*)+EBX+offset(*x*) but we can rearrange this since addition is commutative.

You can also define two-dimensional arrays within a record. Accessing elements of such arrays is no different than any other two-dimensional array other than the fact that you must specify the array's field name as the base address for the array. E.g.,

```
type
    RecW2DArray:
        record
            intField: int32;
            aField: int32[4,5];
                .
                .
                .
        endrecord;

static
    recVar: RecW2DArray;
        .
        .
        .
    // Access element [i,j] of the aField field using Row-major ordering:

    mov( i, ebx );
    intmul( 5, ebx );
    add( j, ebx );
    mov( recVar.aField[ ebx*4 ], eax );
        .
        .
        .
```

The code above uses the standard row-major calculation to index into a 4x5 array of double words. The only difference between this example and a stand-alone array access is the fact that the base address is *recVar.aField*.

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.

If you have an array of records and one of the fields of that record type is an array, you must compute the indexes into the arrays independently of one another and then use the sum of these indexes as the ultimate index. The following example demonstrates how to do this:

```
type
    recType:
        record
            arrayField: dword[4,5];
            << Other Fields >>
        endrecord;

static
    aryOfRecs: recType[3,3];
        .
        .
        .
    // Access aryOfRecs[i,j].arrayField[k,l]:

    intmul( 5, i, ebx );               // Computes index into aryOfRecs
    add( j, ebx );                     //  as (i*5 +j)*@size( recType ).
    intmul( @size( recType ), ebx );

    intmul( 3, k, eax );               // Computes index into aryOfRecs
    add( l, eax );                     //  as (k*3 + j) (*4 handled later).

    mov( aryOfRecs.arrayField[ ebx + eax*4 ], eax );
```

Note the use of the base plus scaled indexed addressing mode to simplify this operation.

5.6 Controlling Field Offsets Within a Record

By default, whenever you create a record, HLA automatically assigns 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. HLA 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;
```

As you're well aware by now, 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.

Generally, you will not use HLA's ability to specify the starting field offset when creating your own record types. Instead, this feature finds most of its use when you are mapping an HLA data type over the top of some other predefined data type in memory (strings are a good example, but there are many other examples as well).

5.7 Aligning Fields Within a Record

To achieve maximum performance in your programs, or to ensure that HLA's 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 to do this, the same way you would use ALIGN in the STATIC declaration section of your program. 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. Note that HLA inserts three bytes of padding between *c* and *d* and it inserts one byte of padding between *b* and *w*. It goes without saying that you should never assume that this padding is present. If you want to use those extra bytes, then declare fields for them.

Note that specifying alignment within a record declaration does not guarantee that the field will be aligned on that boundary in memory; it only ensures that the field's offset is aligned on the specified boundary. If a variable of type *PaddedRecord* starts at an odd address in memory, then the *d* field will also start at an odd address (since any odd address plus four is an odd address). If you want to ensure that the fields are aligned on appropriate boundaries in memory, you must also use the ALIGN directive before variable declarations of that record type, e.g.,


```
static
        .
        .
        .
    align(4);
    PRvar: PaddedRecord;
```

The value of the ALIGN operand should be an even value that is evenly divisible by the largest ALIGN expression within the record type (four is the largest value in this case, and it's already evenly divisible by two).

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;
```

5.8 Pointers to Records

During execution, your program may refer to structure objects directly or indirectly using a pointer. When you use a pointer to access fields of a structure, you must load one of the 80x86's 32-bit registers with the address of the desired record. Suppose you have the following variable declarations (assuming the *Object8* structure from "Arrays/Records as Record Fields" on page 487):

```
static
    Cube:      Object8;
    CubePtr:   pointer to Object8 := &Cube;
```

CubePtr contains the address of (i.e., it is a pointer to) the *Cube* object. To access the *Color* field of the *Cube* object, you could use an instruction like "mov(Cube.Color, eax);". When accessing a field via a pointer you need to load the address of the object into a 32-bit register such as EBX. The instruction "mov(CubePtr EBX);" will do the trick. After doing so, you can access fields of the *Cube* object using the [EBX+offset] addressing mode. The only problem is "How do you specify which field to access?" Consider briefly, the following *incorrect* code:

```
        mov( CubePtr, ebx );
        mov( [ebx].Color, eax );      // This does not work!
```

There is one major problem with the code above. Since field names are local to a structure and it's possible to reuse a field name in two or more structures, how does HLA determine which offset *Color* represents? When accessing structure members directly (.e.g., "mov(Cube.Color, EAX);") there is no ambiguity since *Cube* has a specific type that the assembler can check. "[EBX]", on the other hand, can point at *anything*. In particular, it can point at any structure that contains a *Color* field. So the assembler cannot, on its own, decide which offset to use for the *Color* symbol.

HLA resolves this ambiguity by requiring that you explicitly supply a type. To do this, you must coerce "[EBX]" to type *Cube*. Once you do this, you can use the normal dot operator notation to access the *Color* field:

```
        mov( CubePtr, ebx );
        mov( (type Cube [ebx]).Color, eax );
```

By specifying the record name, HLA knows which offset value to use for the *Color* symbol.

If you have a pointer to a record and one of that record's fields is an array, the easiest way to access elements of that field is by using the base plus indexed addressing mode. To do so, you just load the pointer to the record into one register and compute the index into the array in a second register. Then you combine these two registers in the address expression. In the example above, the *Pts* field is an array of eight *point* objects. To access field *x* of the i^{th} element of the *Cube.Pts* field, you'd use code like the following:

```
mov( CubePtr, ebx );
intmul( @size( point ), i, esi );   // Compute index into point array.
mov( (type Object8 [ebx]).Pts.x[ esi*4 ], eax );
```

As usual, the index appears after all the field names.

If you use a pointer to a particular record type frequently in your program, typing a coercion operator like "(type Object8 [ebx])" can get old pretty quick. One way to reduce the typing needed to coerce EBX is to use a TEXT constant. For example, consider the following statement in a program:

```
const
    O8ptr: text := "(type Object8 [ebx])";
```

With this statement at the beginning of your program you can use *O8ptr* in place of the type coercion operator and HLA will automatically substitute the appropriate text. With a text constant like the above, the former example becomes a little more readable and writable:

```
mov( CubePtr, ebx );
intmul( @size( point ), i, esi );   // Compute index into point array.
mov( O8Ptr.Pts.x[ esi*4 ], eax );
```

5.9 Unions

A record definition assigns different offsets to each field in the record according to the size of those fields. This behavior is quite similar to the allocation of memory offsets in a VAR or STATIC section. HLA provides a second type of structure declaration, the UNION, that does not assign different addresses to each object; instead, each field in a UNION declaration has the same offset – zero. The following example demonstrates the syntax for a UNION declaration:

```
type
    unionType:
        union
            << fields (syntactically identical to record declarations) >>
        endunion;
```

You access the fields of a UNION exactly the same way you access the fields of a record: using dot notation and field names. The following is a concrete example of a UNION type declaration and a variable of the UNION type:

```
type
    numeric:
        union
            i: int32;
            u: uns32;
            r: real64;
        endunion;
            .
            .
            .
static
    number: numeric;
            .
            .
```



```
        .
    mov( 55, number.u );
        .
        .
        .
    mov( -5, number.i );
        .
        .
        .
    stdout.put( "Real value = ", number.r, nl );
```

The important thing to note about UNION objects is that all the fields of a UNION have the same offset in the structure. In the example above, the *number.u*, *number.i*, and *number.r* fields all have the same offset: zero. Therefore, the fields of a UNION overlap one another in memory; this is very similar to the way the 80x86 eight, sixteen, and thirty-two bit registers overlap one another. Usually, access to the fields of a UNION are mutually exclusive; that is, you do not manipulate separate fields of a particular UNION variable concurrently because writing to one field overwrite's the other fields. In the example above, any modification of number.u would also change *number.i* and *number.r*.

Programmers typically use UNIONs for two different reasons: to conserve memory or to create aliases. Memory conservation is the intended use of this data structure facility. To see how this works, let's compare the *numeric* UNION above with a corresponding record type:

```
type
    numericRec:
        record
            i: int32;
            u: uns32;
            r: real64;
        endrecord;
```

If you declare a variable, say *n*, of type *numericRec*, you access the fields as *n.i*, *n,u*, and *n.r*; exactly as though you had declared the variable to be type *numeric*. The difference between the two is that *numericRec* variables allocate separate storage for each field of the record while numeric objects allocate the same storage for all fields. Therefore, *@size(numericRec)* is 16 since the record contains two double word fields and a quad word (*real64*) field. *@size(numeric)*, however, is eight. This is because all the fields of a UNION occupy the same memory locations and the size of a UNION object is the size of the largest field of that object (see Figure 5.2).

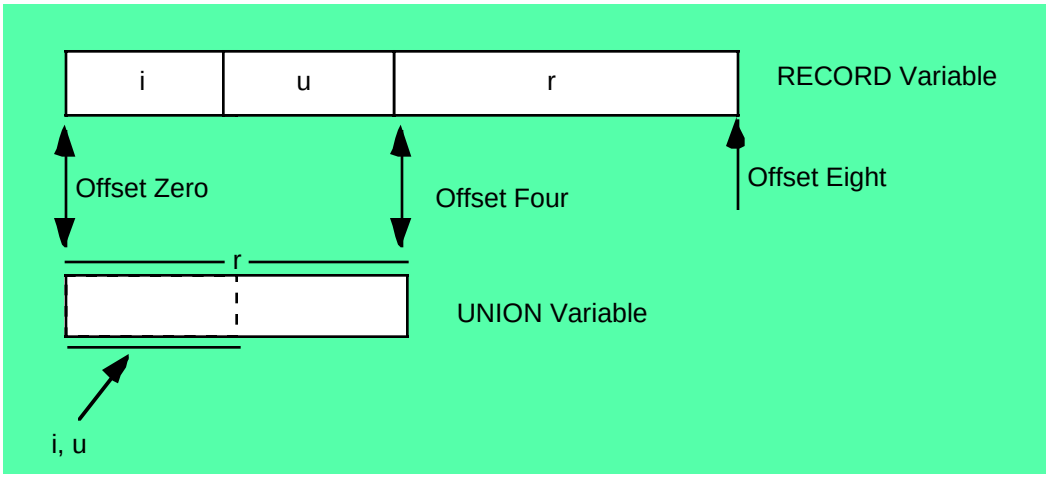

Figure 5.2 Layout of a UNION versus a RECORD Variable

In addition to conserving memory, programmers often use UNIONs to create aliases in their code. As you may recall, an alias is a different name for the same memory object. Aliases are often a source of confusion in a program so you should use them sparingly; sometimes, however, using an alias can be quite convenient. For example, in some section of your program you might need to constantly use type coercion to refer to an object using a different type. Although you can use an HLA TEXT constant to simplify this process, another way to do this is to use a UNION variable with the fields representing the different types you want to use for the object. As an example, consider the following code:

```
type
    CharOrUns:
        union
            c:char;
            u:uns32;
        endrecord;

static
    v:CharOrUns;
```

With a declaration like the above, you can manipulate an *uns32* object by accessing *v.u*. If, at some point, you need to treat the L.O. byte of this *uns32* variable as a character, you can do so by simply accessing the *v.c* variable, e.g.,

```
mov( eax, v.u );
stdout.put( "v, as a character, is '", v.c, "'" nl );
```

You can use UNIONs exactly the same way you use RECORDs in an HLA program. In particular, UNION declarations may appear as fields in RECORDs, RECORD declarations may appear as fields in UNIONs, array declarations may appear within UNIONs, you can create arrays of UNIONs, etc.

5.10 Anonymous Unions

Within a RECORD declaration you can place a UNION declaration without specifying a fieldname for the union object. The following example demonstrates the syntax for this:

```
type
    HasAnonUnion:
        record
```



```
            r:real64;
            union
                u:uns32;
                i:int32;
            endunion;
            s:string;
        endrecord;

static
    v: HasAnonUnion;
```

Whenever an anonymous union appears within an RECORD you can access the fields of the UNION as though they were direct fields of the RECORD. In the example above, for example, you would access *v's u* and *i* fields using the syntax "v.u" and "v.i", respectively. The *u* and *i* fields have the same offset in the record (eight, since they follow a *real64* object). The fields of *v* have the following offsets from *v's* base address:

```
v.r     0
v.u     8
v.i     8
v.s    12
```

@size(v) is 16 since the *u* and *i* fields only consume four bytes between them.

Warning: HLA gets confused if you attempt to create a record constant when that record has anonymous unions (HLA doesn't allow UNION constants). So don't create record constants of a record if that record contains anonymous unions as fields.

HLA also allows anonymous records within unions. Please see the HLA documentation for more details, though the syntax and usage is identical to anonymous unions within records.

5.11 Variant Types

One big use of UNIONs in programs is to create *variant* types. A variant variable can change its type dynamically while the program is running. A variant object can be an integer at one point in the program, switch to a string at a different part of the program, and then change to a real value at a later time. Many very high level language systems use a dynamic type system (i.e., variant objects) to reduce the overall complexity of the program; indeed, proponents of many very high level languages insist that the use of a dynamic typing system is one of the reasons you can write complex programs in so few lines. Of course, if you can create variant objects in a very high level language, you can certainly do it in assembly language. In this section we'll look at how we can use the UNION structure to create variant types.

At any one given instant during program execution a variant object has a specific type, but under program control the variable can switch to a different type. Therefore, when the program processes a variant object it must use an IF statement or SWITCH statement to execute a different sequence of instructions based on the object's current type. Very high level languages (VHLLs) do this transparently. In assembly language you will have to provide the code to test the type yourself. To achieve this, the variant type needs some additional information beyond the object's value. Specifically, the variant object needs a field that specifies the current type of the object. This field (often known as the *tag* field) is a small enumerated type or integer that specifies the type of the object at any given instant. The following code demonstrates how to create a variant type:

```
type
    VariantType:
        record
            tag:uns32;  // 0-uns32, 1-int32, 2-real64
            union
                u:uns32;
                i:int32;
                r:real64;
```



```
            endunion;
        endrecord;

static
    v:VariantType;
```

The program would test the *v.tag* field to determine the current type of the *v* object. Based on this test, the program would manipulate the v.*i*, *v.u*, or *v.r* field.

Of course, when operating on variant objects, the program's code must constantly be testing the tag field and executing a separate sequence of instructions for uns32, int32, or real64 values. If you use the variant fields often, it makes a lot of since to write procedures to handle these operations for you (e.g., *vadd*, *vsub*, *vmul*, and *vdiv*). Better yet, you might want to make a class out of your variant types. For details on this, see the chapter on Classes appearing later in this text.

5.12 Namespaces

One really nice feature of RECORDs and UNIONs is that the field names are local to a given RECORD or UNION declaration. That is, you can reuse field names in different RECORDs or UNIONs. This is an important feature of HLA because it helps avoid *name space pollution*. Name space pollution occurs when you use up all the "good" names within the current scope and you have to start creating non-descriptive names for some object because you've already used the most appropriate name for something else. Because you can reuse names in different RECORD/UNION definitions (and you can even reuse those names outside of the RECORD/UNION definitions) you don't have to dream up new names for the objects that have less meaning. We use the term *namespace* to describe how HLA associates names with a particular object. The field names of a RECORD have a namespace that is limited to objects of that record type. HLA provides a generalization of this namespace mechanism that lets you create arbitrary namespaces. These namespace objects let you shield the names of constants, types, variables, and other objects so their names do not interfere with other declarations in your program.

An HLA NAMESPACE section encapsulates a set of generic declarations in much the same way that a RECORD encapsulates a set of variable declarations. A NAMESPACE declaration takes the following form:

```
namespace name;

    << declarations >>

end name;
```

The *name* identifier provides the name for the NAMESPACE. The identifier after the END clause must exactly match the identifier after NAMESPACE. You may have several NAMESPACE declarations within a program as long as the identifiers for the name spaces are all unique. Note that a NAMESPACE declaration section is a section unto itself. It does not have to appear in a TYPE or VAR section. A NAMESPACE may appear anywhere one of the HLA declaration sections is legal. A program may contain any number of NAMESPACE declarations; in fact, the name space identifiers don't even have to be unique as you will soon see.

The declarations that appear between the NAMESPACE and END clauses are all the standard HLA declaration sections except that you cannot nest name space declarations. You may, however, put CONST, VAL, TYPE, STATIC, READONLY, STORAGE, and VAR sections within a namespace[3]. The following code provides an example of a typical NAMESPACE declaration in an HLA program:

```
namespace myNames;
```

3. Procedure and macro declarations, the subjects of later chapters, are also legal within a name space declaration section.


```
    type
        integer: int32;

    static
        i:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;
```

To access the fields of a name space you use the same dot notation that records and unions use. For example, to access the fields of *myNames* outside of the name space you'd use the following identifiers:

```
myNames.integer - A type declaration equivalent to int32.
myNames.i - An integer variable (int32).
myNames.j - An uns32 variable.
myNames.pi - A real64 constant.
```

This example also demonstrates an important point about NAMESPACE declarations: within a name space you may reference other identifiers in that same NAMESPACE declaration without using the dot notation. For example, the *i* field above uses type *integer* from the *myNames* name space without the "mynames." prefix.

What is not obvious from the example above is that NAMESPACE declarations create a clean symbol table whenever you open up a declaration. The only external symbols that HLA recognizes in a NAMESPACE declaration are the predefined type identifiers (e.g., int32, uns32, and char). HLA does not recognize any symbols you've declared outside the NAMESPACE while it is processing your namespace declaration. This creates a problem if you want to use symbols outside the NAMESPACE when declaring other symbols inside the NAMESPACE. For example, suppose the type *integer* had been defined outside *myNames* as follows:

```
type
    integer: int32;

namespace myNames;


    static
        i:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;
```

If you were to attempt to compile this code, HLA would complain that the symbol *integer* is undefined. Clearly *integer* is defined in this program, but HLA hides all external symbols when creating a name space so that you can reuse (and redefine) those symbols within the name space. Of course, this doesn't help much if you actually want to use a name that you've defined outside *myNames* within that name space. HLA provides a solution to this problem: the *@global:* operator. If, within a name space declaration section, you prefix a name with "@global:" then HLA will use the global definition of that name rather than the local definition (if a local definition even exists). To correct the problem in the previous example, you'd use the following code:

```
type
    integer: int32;
```



```
namespace myNames;


    static
        i:@global:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;
```

With the *@global:* prefix, the *i* variable will be type *int32* even if a different declaration of integer appears within the *myNames* name space.

You cannot nest NAMESPACE declarations[4]. However, you can have multiple NAMESPACE declarations in the same program that use the same name space identifier, e.g.,

```
namespace ns;

    << declaration group #1 >>

end ns;
    .
    .
    .
namespace ns;

    << declaration group #2 >>

end ns;
```

When HLA encounters a second NAMESPACE declaration for a given identifier, it simply appends the declarations in the second group to the end of the symbol list it created for the first group. Therefore, after processing the two NAMESPACE declarations, the *ns* name space would contain the set of all symbols you've declared in both the name space blocks.

Perhaps the most common use of name spaces is in library modules. If you create a set of library routines to use in various projects or distribute to others, you have to be careful about the names you choose for your functions and other objects. If you use common names like *get* and *put*, the users of your module will complain when your names collide with theirs. An easily solution is to put all your code in a NAMESPACE block. Then the only name you have to worry about is the name of the NAMESPACE itself. This is the only name that will collide with other users' code. That can happen, but it's much less likely to happen than if you don't use a name space and your library module introduces dozens, if not hundreds, of new names into the global name space[5]. The HLA Standard Library provides many good examples of name spaces in use. The HLA Standard Library defines several name spaces like *stdout*, *stdin*, *str*, *cs*, and *chars*. You refer to functions in these name spaces using names like *stdout.put*, *stdin.get*, *cs.intersection*, *str.eq*, and *chars.toUpper*. The use of name spaces in the HLA Standard Library prevents conflicts with similar names in your own programs.

5.13 Putting It All Together

One of the more amazing facts about programmer psychology is the fact that a high level language programmer would refuse to use a high level language that doesn't support records or structures; then that same programmer won't bother to learn how to use them in assembly language (all the time, grumbling about their

4. There really doesn't seem to be a need to do this; hence its omission from HLA.
5. The global name space is the global section of your program.

absence). You use records in assembly language for the same reason you use them in high level languages. Given that most programmers consider records and structure essential in high level languages, it is surprising they aren't as concerned about using them in assembly language.

This short chapter demonstrates that it doesn't take much effort to master the concept of records in an assembly language program. Taken together with UNIONs and NAMESPACEs, RECORDs can help you write HLA programs that are far more readable and easier to understand. Therefore, you should use these language features as appropriate when writing assembly code.

