**Reading COBOL Layouts**

|  |
| --- |
| This tutorial on how to read a COBOL layout was written specifically for our customers who have had a conversion performed at Disc Interchange and have received a COBOL layout with the data.  It is intended to give you enough information to read most simple layouts.  It does not cover all topics or everything you would find in a complex layout, and it is intended to explain COBOL layouts only so you can use your converted data, not so you can write COBOL programs.  This article begins here: [Reading COBOL Layouts](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts.html) where you will also find a topic index. |

**Part 1: COBOL Basics**

We will look at some basic COBOL rules, then at some examples, expanding on the concepts as we go.  Newly introduced terms are listed in **bold.**

**Contents of this section:**

|  |
| --- |
| **Need to convert COBOL files?**[Request a COBOL quote](http://www.3480-3590-data-conversion.com/quote2.html) **That's our business!** |

   [Record Layouts](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-1.html#Record Layouts)   
   [Fields and the PIC Clause](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-1.html#Fields and the PIC Clause)   
   [Filler](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-1.html#Filler)   
   [Special Formatting Characters](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-1.html#Special Formatting Characters)   
   [Columns, Line Numbers, and Comments](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-1.html#Columns, Line Numbers & Comments)   
   [Levels and Groups](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-1.html#Levels and Groups)   
   [COBOL 's 66 and 88 Levels](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-1.html#COBOL 66 & 88 Levels)

**Record Layouts**

A **record layout**is a description of all the individual **fields** that comprise each **record** in the data **file**. COBOL layouts follow specific rules. Since we are not teaching you how to program in COBOL, we will only discuss the rules you need to know to *read* layouts.

The layout specifies at-least the name of each field, its type, size, and position in the record.  A layout may give a detailed description of the use of each field and the values found in it, but that information is often contained in the **data dictionary.**  A COBOL layout usually pertains to a single disk or tape file, as opposed to a **table** within a **database**.

**Fields and the PIC clause**

The lowest level data item in a COBOL layout is a field, also called an **elementary item**.  Several fields can be associated to form a **group**.  All the fields together form a **record**.

A COBOL layout is comprised of a line for each **field** or **group**.  A COBOL field definition gives the **level** (discussed later), field name, and a "picture", or **PIC** clause, which tells you the **data type** or **data category** of the field, and its size.  The three data types you are likely to see are:

1. "**A**" for alpha (A-Z, a-z, and space only).
2. "**9**" for a numeric field (numbers 0-9, but no letters).
3. "**X**" for any character, (including binary).

For example, the following field (elementary item) is called ZIP-CODE and is 5 digits wide, as specified by the five 9s.  i.e., the "picture" of the field is "99999".

  05  ZIP-CODE      PIC 99999.

This could also be written:

  05  ZIP-CODE      PIC 9(5).

Where the 9 means the field type is numeric, like the first example, and the (5) says there are five digits.  The 9(5) and the 99999 are identical field specifications. The parentheses are usually used when it makes the definition shorter or clearer, as in 9(11) vs: 99999999999.  The period at the end separates this field definition from the next one.

A character field such as last name could be written:

   05  LAST-NAME     PIC A(15).

Meaning it's a 15 character alphabetic field.  But it's actually more common to see character fields specified as PIC X, like:

   05  LAST-NAME     PIC X(15).

PIC X allows *any* character, including numbers, punctuation, and binary codes.

Like the numeric example above, a PIC X field specification could be written as either multiple Xs or a count in parentheses, like these two identical field specifications:

   05  LAST-NAME     PIC X(15).

   05  LAST-NAME     PIC XXXXXXXXXXXXXXX.

Although not commonly seen in COBOL files, you can mix types in a field.  For example,

   05  ZIP-PLUS-9      PIC 99999X9999.

permits a dash (or anything) between ZIP and ZIP+4, like 01886-2001.

A decimal point in a PIC, like "PIC  999.99" separates the integer portion from the decimal portion.  This is discussed in more detail later, along with **implied decimal**.

Let's practice one more, just to get the point across.  The following are different ways of specifying the *same thing:*

   05  AMOUNT          PIC 999.99.

   05  AMOUNT          PIC 9(3).9(2).

   05  AMOUNT          PIC 9(3).99.

   05  AMOUNT          PIC 999.9(2).

**Filler**

There is a special type of COBOL field called **FILLER**.  This reserves space in a COBOL record, commonly for future expansion or to fill a gap created by a **redefined field**.  FILLER is a **reserved word**, and you can have as many FILLER fields in a record as you want -- the name does not have to be unique as field names generally must be.

Filler can also be used to create a field, or place holder, that you will never need to refer to by name, so you might find it contains actual data, not just blank space.

It's also common for a vendor to use fields for some internal purpose, for example as a key field, but to mark those fields as FILLER when the data is sent outside the company. So FILLER fields can contain anything, including binary data. You should not expect them to be neatly filled with spaces.

**Special Formatting Characters**

There are a number of special characters that cause specific actions on the data, such as leading zeros or spaces, floating signs, leading or trailing signs, decimal points, etc.  We will mention only a few common ones:

1. A **literal** in a field causes that character to appear in that location. For example,

   05  ZIP-PLUS-9     PIC 99999-9999.

specifies a field with five digits, a dash, and four more digits.  The dash is not part of the variable data -- it is a *literal*character.

2. A decimal point in a numeric field does *two*things: it places an actual decimal point into the file, and determines the location of the decimal for calculations.  The following field is *six* bytes wide and has a "**real decimal**" in the file:

   05  AMOUNT      PIC 999.99.

If you view a record containing the value 123.45 in this field, you will see "123.45"

3. A "**V**" in the PIC clause specifies the location of an **implied decimal**.  This is discussed later, in the section on numeric fields.  The following field is *five* bytes wide and has an "implied decimal" at the location of the V:

   05  AMOUNT      PIC 999V99.

If you view a record containing the value 123.45 in this field, you will see "12345" .

4. A minus sign, "-", reserves a byte in the record for an actual sign, and puts a "-" in negative values, and a space in positive values.

5. Similarly, a "+" in the PIC puts a "-" in negative values and a "+" in positive values.  See the section below on "**signed fields**" for the representation of PIC S9 fields.

6. A "**P**" in a PIC clause scales the value.  This is seldom seen, so we will be brief, via two examples: 

|  |  |
| --- | --- |
| PIC 999PPP. | The three 9s cause this field to be three bytes in size, and the three Ps scale it UP by 1000. If the field contains the digits 123, the actual value represented is 123,000. |
| PIC PPP999. | This scales the value DOWN.  If the field contains the digits 123 the actual value is 0.000123 |

**Columns, Line Numbers, and Comments**

Columns 1-6 in most COBOL layouts are ignored by the compiler, as is everything after column 72.  You will often find **line numbers** or other **comments** (such as when a field was added or changed, or where it originated) in these columns.  These may be useful to you in finding your way around a large layout; just be aware they are ignored by the compiler.

COBOL layouts are divided into "**areas**", and there are many rules for what data may be found in which area, but one you should remember is that an asterisk, \*, in column 7, the "**indicator area**" turns the entire line into a comment, which is ignored by the COBOL compiler.  Even if that line contains a field specification, it will be ignored if there is an \* in column 7.

There are variations on COBOL layouts that discard columns 1-6, shifting the entire layout left.  And, some printed documentation may not show these columns.  You can usually find your place in the layout from the 01 level, which normally starts in column 8.  All other levels should start in column 12 or above.

A COBOL field definition does not need to be entirely on one line.  A line-ending has no significance to the compiler; it's the period at the end that's the COBOL **separator**, not the carriage-return.

**Levels and Groups**

COBOL layouts have **levels,**from level 01 to level 49.  These levels tell the COBOL compiler how to associate, or group, fields in the record.  Level 01 is a special case, and is reserved for the record level;  the 01 level is the *name of the record*. Levels from 02 to 49 are all "equal" (level 2 is no more significant than level 3), but there is a hierarchy to the structure.  Any field listed in a lower level (higher number) is subordinate to a field or group in a higher level (lower number).  For example, LAST-NAME and FIRST-NAME in the example below are part of, or belong to, the group CUSTOMER-NAME, as can be seen by the level numbers of 05 and 10.

   05  CUSTOMER-NAME.

       10  LAST-NAME           PIC X(15).

       10  FIRST-NAME          PIC X(8).

Notice that CUSTOMER-NAME does not have a PIC, since it's a **group**, not a field.  Also notice that the two fields within the group are at a lower level, level 10, than the 05 group.  Lower levels are normally indented further for clarity, but this is not required, and in fact the compiler doesn't care.

For the rest of this tutorial we will use levels 05, 10, and 15 to be consistent.  Just remember these choices are arbitrary; we could have used 02, 03, and 04, or any other numbers between 02 and 49.

There can be many levels.  Here is a brief example of a record with three levels:

   01  MAILING-RECORD.

       05  COMPANY-NAME            PIC X(30).

       05  CONTACTS.

           10  PRESIDENT.

               15  LAST-NAME       PIC X(15).

               15  FIRST-NAME      PIC X(8).

           10  VP-MARKETING.

               15  LAST-NAME       PIC X(15).

               15  FIRST-NAME      PIC X(8).

           10  ALTERNATE-CONTACT.

               15  TITLE           PIC X(10).

               15  LAST-NAME       PIC X(15).

               15  FIRST-NAME      PIC X(8).

       05  ADDRESS                 PIC X(15).

       05  CITY                    PIC X(15).

       05  STATE                   PIC XX.

       05  ZIP                     PIC 9(5).

Most of the fields in this record (company, address, city, state, zip) are simple fields that need no comment. But there are some interesting things about the contact fields:

There is a group called CONTACTS at the 05 level.  Within this group are three 10 level groups.  The first one is PRESIDENT, and within this group are the LAST-NAME and FIRST-NAME fields for the president.  So far this is similar to the previous example, with one more level.  This group is 23 bytes (15 + 8).

Next we have a group to contain the name of the VP of Marketing. This group is also 23 bytes.  Notice it uses the same field names, LAST-NAME and FIRST-NAME, as used in the president's group.  Although this isn't commonly seen, it is permitted in COBOL. They are considered different fields because they are within different groups.  In COBOL you distinguish them by referring to "LAST-NAME OF PRESIDENT" for the president's name, and "LAST-NAME OF VP-MARKETING" for the name of the VP of Marketing.

The last group in the CONTACTS group is for "alternate contacts".  This one contains a field called TITLE which contains the title of the alternate contact (e.g., CEO). Like the others, it contains LAST-NAME and FIRST-NAME fields. This group is 33 bytes. 

**COBOL's 66 and 88 Levels**

These two levels have special meaning.  The 66 level assigns an alternate name to a field or group. It doesn't add a new field to the record, it just assigns an alternate name to an existing field.  You are not likely to see level 66.

You are likely to see the 88 level, though.  The 88 level simply equates a value with a name.  Here's a simple example:

   05  SEX                   PIC X.

       88  MALE     VALUE "M".

       88  FEMALE   VALUE "F".

This equates the value "M" with "MALE", and the value "F" with "FEMALE" for the field SEX.  (This allows your COBOL program to, for example, test IF MALE rather than having to say IF SEX IS EQUAL TO "M").  Since we are not teaching COBOL programming, this is incidental to us, but here's what is important to know about the 88 level:

1. The 88 level does not define a field, and does not take space in the record; it is merely a value definition.
2. The 88 level does not limit the possible codes to only those listed.  There *could*be other values used in that field;  M and F are not the only values you might find.  (Although a good layout *will* list them all.)   In this case there might be a "U" (unknown), or a blank.
3. If the layout is complete, this is a handy list of the values you can expect to find in this field.  Sometimes it's all you have to go on.

88 levels may specify multiple values, or a range of values, such as:

       88  ODD-NUMBERS  VALUE 1, 3, 5, 7, 9.

       88  PRE-SCHOOL   VALUE 0 THROUGH 4.

**Part 2: Simple COBOL Layouts**

This section describes a very simple COBOL layout, then introduces the concept of groups.

**Contents of this section:**

|  |
| --- |
| **Need to convert COBOL files?**[Request a COBOL quote](http://www.3480-3590-data-conversion.com/quote2.html) **That's our business!** |

   [The  Simplest Example](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-2.html#The Simplest Layout)   
   [Groups](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-2.html#Groups)

Let's start with a simple layout and add features as we go.

**The Simplest Example**

This is a very simple COBOL layout for a customer list.  The name of the record is "CUSTOMER-RECORD".  All the fields are at the same level (05), there are no groups.

   01  CUSTOMER-RECORD.

       05  LAST-NAME               PIC X(15).

       05  FIRST-NAME              PIC X(8).

       05  STREET-ADDRESS          PIC X(20).

       05  CITY                    PIC X(17).

       05  STATE                   PIC XX.

       05  ZIP-CODE                PIC 9(5).

       05  FILLER                  PIC X(10).

Notice a few things:

1. A field name can have dashes between words, but not spaces.
2. The 01 level specifies the name of the record, and does not have a PIC (it's composed of all the fields that follow).
3. All the fields in this record are at the 05 level, so none are subordinate to any others; they are all just one-after-another in the file.
4. The 05 level is arbitrary; it could have been anything from 02 to 49.
5. There is a FILLER field at the end of the record to reserve space for future expansion.
6. The record size is the sum of the fields, or 15 + 8 + 20 + 17 + 2 + 5 + 10 = 77 bytes.
7. The starting position of each field is not explicitly given, but is determined by the sum of all the fields before it.

**Groups**

Now let's add a group called CUSTOMER-NAME, which will be composed of the fields LAST-NAME and FIRST-NAME:

   01  CUSTOMER-RECORD.

       05  CUSTOMER-NAME.

           10  LAST-NAME           PIC X(15).

           10  FIRST-NAME          PIC X(8).

       05  STREET-ADDRESS          PIC X(20).

       05  CITY                    PIC X(17).

       05  STATE                   PIC XX.

       05  ZIP-CODE                PIC 9(5).

       05  FILLER                  PIC X(10).

This record has *exactly* the same field positions as the previous layout.  The group CUSTOMER-NAME at the 05 level is composed of all the fields subordinate to it, down to the next 05 field.  i.e., it's composed of  LAST-NAME and FIRST-NAME.  CUSTOMER-NAME is a *group*, not a field, and therefore does not have (and cannot have) a PIC, and the line that reads "05 CUSTOMER-NAME" does not, by itself, reserve any space in the record.

Groups are handy for a number of reasons.  You can operate on all the fields in a group by referring to the name of the group.  If you reference CUSTOMER-NAME in the above layout, it will implicitly include all lower levels, LAST-NAME and FIRST-NAME in this case.

You can also have groups within a group. The following example, used in part 1 of this article is an example:

   01  MAILING-RECORD.

       05  COMPANY-NAME            PIC X(30).

       05  CONTACTS.

           10  PRESIDENT.

               15  LAST-NAME       PIC X(15).

               15  FIRST-NAME      PIC X(8).

           10  VP-MARKETING.

               15  LAST-NAME       PIC X(15).

               15  FIRST-NAME      PIC X(8).

           10  ALTERNATE-CONTACT.

               15  TITLE           PIC X(10).

               15  LAST-NAME       PIC X(15).

               15  FIRST-NAME      PIC X(8).

       05  ADDRESS                 PIC X(15).

       05  CITY                    PIC X(15).

       05  STATE                   PIC XX.

       05  ZIP                     PIC 9(5).

Notice the group CONTACTS is composed of three subordinate groups: PRESIDENT, VP-MARKETING, and ALTERNATE-CONTACT. Notice, too, that the group CONTACTS contains no elementary items (fields), only groups.  It could have contained elementary items, such as the following fragment:

       05  CONTACTS.

           10 SOURCE-OF-NAMES      PIC XX.

           10  PRESIDENT.

               15  LAST-NAME       PIC X(15).

               15  FIRST-NAME      PIC X(8).

           10  VP-MARKETING.

               15  LAST-NAME       PIC X(15).

               15  FIRST-NAME      PIC X(8).

           10  ALTERNATE-CONTACT.

               15  TITLE           PIC X(10).

               15  LAST-NAME       PIC X(15).

               15  FIRST-NAME      PIC X(8).

As noted in Part 1 of this article, it is legal to have multiple fields called LAST-NAME and FIRST-NAME, because they are in different groups.  Although it's legal, it's still wise to make each field name unique, and you are more likely to see them prefixed or suffixed, like PRES-LAST-NAME, VP-LAST-NAME and ALT-LAST-NAME.

## Part 3: Redefined Fields

This section introduces the concept of Redefined Fields and explains their use.  There is a simpler introduction to redefined fields and records in the article [Redefined Fields & Records](http://www.3480-3590-data-conversion.com/article-redefined.html)

**Contents of this section:**

|  |
| --- |
| **Problems with Redefined fields?**[Request a COBOL quote](http://www.3480-3590-data-conversion.com/quote2.html) **We can help!** |

   [The Concept of Redefined Fields](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-3.html#The Concept of Redefined Fields)   
   [Determining Which Definition to Use](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-3.html#Determining Which Definition to Use)   
   [Other Reasons for Redefining Fields](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-3.html#Other Reasons for Redefined Fields)   
   [Ignoring Redefined Fields](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-3.html#Ignoring Redefined Fields)

### The Concept of Redefined Fields

The area occupied by a field in a COBOL layout can be re-used for different data by **redefining** the original field. This does exactly what it says -- it redefines the space used by a field for another use.

In the example below, the field PO-BOX redefines the field STREET-ADDRESS.

   05  STREET-ADDRESS                    PIC X(20).

   05  PO-BOX REDEFINES STREET-ADDRESS   PIC X(20).

Notice these are both 20 bytes, and they *both* occupy the *same* 20 bytes in the record, because the second definition *redefines* the first one.  When you add the field sizes to determine the starting column of each field, be sure you don't count the size of the redefining fields. To do so would count those fields twice.

Fields and groups can both be redefined by other fields or groups.  That is, a field can redefine another field or group, and a group can redefine a field or another group.

In the example below, the field PO-BOX redefines the group STREET-ADDRESS.

   05  STREET-ADDRESS.

       10  ADDRESS-LINE-1               PIC X(20).

       10  ADDRESS-LINE-2               PIC X(20).

   05  PO-BOX REDEFINES STREET-ADDRESS  PIC X(40).

Again, notice they are both the same size --  40 bytes.

In the example below, the group RURAL-ADDRESS redefines the group STREET-ADDRESS.

   05  STREET-ADDRESS.

       10  ADDRESS-LINE-1         PIC X(20).

       10  ADDRESS-LINE-2         PIC X(20).

   05  RURAL-ADDRESS REDEFINES STREET-ADDRESS.

       10  RURAL-ROUTE            PIC 9(3).

       10  RR-BOX-NUMBER          PIC X(10).

       10  FILLER                 PIC X(27).

The total size of the fields in both the original group and the redefining group is 40 bytes.  Notice the use of the FILLER to make up the difference. Notice, too, that RURAL-ROUTE is a numeric field redefining an alpha-numeric field.

Also notice that the field sizes in the two groups are not the same. RURAL-ROUTE is three bytes, while ADDRESS-LINE-1 is 20 bytes.  This, and the filler at the end, precludes you from using just the first definition and ignoring the second definition.  (Recall that FILLER fields can contain *any* value, so you can't count on them containing spaces).

Now let's add a redefined field to the layout of the previous section:

   01  CUSTOMER-RECORD.

       05  CUSTOMER-NAME.

           10  LAST-NAME           PIC X(15).

           10  FIRST-NAME          PIC X(8).

       05  COMPANY-NAME REDEFINES CUSTOMER-NAME

                                   PIC X(23).

       05  STREET-ADDRESS          PIC X(20).

       05  CITY                    PIC X(17).

       05  STATE                   PIC XX.

       05  ZIP-CODE                PIC 9(5).

       05  FILLER                  PIC X(10).

COMPANY-NAME is a field that redefines the group CUSTOMER-NAME.  COMPANY-NAME is at the same level, 05, as the field it redefines, and it names the field it redefines, CUSTOMER-NAME.  Notice the field specification can span lines; it's the period that ends it, not the CR-LF. A field or group can be redefined multiple times; there could be a third definition for that area of the record.

In versions of COBOL prior to COBOL 85 the size of a redefining field (COMPANY-NAME) or group had to be the same size or smaller than the field or group it redefined.  If it is smaller it's good practice to add a FILLER field to make up the difference, although this is not mandatory. It is no longer required that the first definition be the largest, although it's still common practice.

Starting with the 1985 COBOL standard; compilers will adjust the size of *all*redefines to make them all the same size.  But the layout r*eally should* specify these sizes to minimize confusion.  Just be aware that a redefines *could* change the size of another field or group at compile time.

To recap what we have in our record above, bytes 1-23 contain either a person's name (last name then first name), or a company name. Bytes 24-43 contain a street address, bytes 44-60 the city, bytes 61-62 the state code, bytes 63-67 the ZIP-CODE, which must be numeric, and bytes 68-77 are FILLER. 

### Determining Which Definition to Use

Our record layout can now accommodate either individual name or company name, but we have no way to know if a particular record has a company name or an individual name.  So we'll add a field called "TYPE-OF-NAME" to indicate which data the record contains -- i.e., which definition to use.

   01  CUSTOMER-RECORD.

       05  TYPE-OF-NAME            PIC X.

           88 PERSON               VALUE 0.

           88 CORPORATION          VALUE 1.

       05  CUSTOMER-NAME.

           10  LAST-NAME           PIC X(15).

           10  FIRST-NAME          PIC X(8).

       05  COMPANY-NAME REDEFINES CUSTOMER-NAME

                                   PIC X(23).

       05  STREET-ADDRESS          PIC X(20).

       05  CITY                    PIC X(17).

       05  STATE                   PIC XX.

       05  ZIP-CODE                PIC 9(5).

       05  FILLER                  PIC X(10).

If TYPE-OF-NAME is set to 0 we know this record contains a person's name, and if it's set to 1 we know it contains a company name.  These values are the binary values 00 hex and 01 hex, so when using this file you will have to deal with the binary data (or have us convert it).

### Other Reasons For Redefining Fields

The reason for the redefined fields above is that we may want to alternate using individual and company names, but sometimes fields get redefined for other reasons. Some possible reasons are:

1. Sometimes a redefined field is just to have two different *views* of the same data.  The date in the example below is always the same data (i.e., it's not *changed* by the redefine), but you can refer to the *entire* date by referencing the field EXPIRATION-DATE, or you can perform calculations on just the year, month, or day by using the redefined fields (note the redefined is numeric).

   05  EXPIRATION-DATE          PIC X(8).

   05  EXPIRATION-DATE-NUMERIC REDEFINES EXPIRATION-DATE.

       10  EXPIRATION-YEAR      PIC 9(4).

       10  EXPIRATION-MONTH     PIC 99.

       10  EXPIRATION-DAY       PIC 99.

2. A program may no longer use a field, and the programmer will re-use that space *for completely new data* with a redefine.  This is safer than just changing the name of the field, since any old programs that might be left around would crash if the old field was simply gone.  In this case there may be no information like the "TYPE-OF-NAME" field above; the newly modified program will simply use the new definition.  That may be okay for the originator of the data, because he knows what he did, but it leaves a new user of the file wondering which definition to use.  The data dictionary may help explain this change.

3. Sometimes the same type of data will be stored two different ways.  For example, an account number may be in alpha-numeric format in some records and comp-3 format in others.  Although that may sound absurd, there may be valid historical reasons for it, and we see it more often than you'd expect.  One company may acquire another and merge the two databases.  The original company may use a 5 character alpha-numeric with a letter prefix, and the new company may use a 9 digit packed field. To convert either could mean major reprogramming, but they will both fit in the original 5 byte field with a redefine.

### Ignoring Redefined Fields

If a redefined field is a different data type than the original definition, such as a binary numeric field redefining a character field, generally you cannot ignore the redefinition.  You will need to treat the records with binary numbers as numeric data and the records with letters as character data.  However, if both the original field and the redefined field are the same data type, such as character fields, you may be able to use the original definition and ignore the redefines.

For instance, in the address example at the beginning of this section, perhaps it doesn't matter to you that the address may be a street address or a Post Office Box -- you will treat both the same way.  In this case you can ignore the redefines.

However, even if the data types are the same, the content may force you to deal with both definitions.  For example, consider two records, one for the individual "Smith      John   " and the other for the company "Disc Interchange   ". If you ignore the redefined issue, and treat the field as the company definition, then "Disc Interchange" will be correct, but the mail to John Smith will be addressed to "Smith       John" (including the spaces). If you treat the fields as an individual's name and put the first name before the last name, then the individual's name will be correct, like "John Smith", but any company name will get scrambled, like "ange Disc Interch". If your application can't deal with this, Disc Interchange can convert the data to a record with *both* individual name fields and company name fields.

**Part 4: Numeric Fields**

This section describes several numeric data types and the handling of signs and decimal points.

**Contents of this section:**

|  |
| --- |
| **Need to convert Numeric fields?**[Request a COBOL quote](http://www.3480-3590-data-conversion.com/quote2.html) **That's our business!** |

   [The "Usage Is" Clause](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-4.html#The Usage is Clause)   
   [Usage is Display](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-4.html#Usage is Display)   
   [Signed Fields](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-4.html#Signed Fields)   
   [Sign is Separate](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-4.html#Sign is Separate)   
   [Computational and Binary Fields](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-4.html#Computational and Binary Fields)   
   [Real Decimal](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-4.html#Real Decimal)   
   [Implied Decimal](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-4.html#Implied Decimal)   
   [Synchronization and Alignment](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-4.html#Synchronization and Alignment)

COBOL has several types of numeric fields. These data types include a "DISPLAY" field, which is composed of characters (the EBCDIC or ASCII characters for 0 - 9), binary fields, packed fields, and floating-point fields. There are also options for a separate + or - sign or a sign overpunch, and for real or implied decimal.  The data type is specified by the "USAGE IS" clause.

**The "USAGE IS" Clause**

There is actually more to the picture statement than we've previously described.  There is a "**USAGE IS**" clause that specifies the type of storage of a numeric field -- "display"**,**binary, or computational.  The full syntax, via an example, is:

   05  ACCOUNT-BALANCE    PIC S9(6)V99 USAGE IS COMPUTATIONAL-3.

This says to store the field in the computational-3 format.  The "usage is" part is optional and generally left off, and "computational" can be abbreviated "COMP", so you will more commonly see this written

   05  ACCOUNT-BALANCE    PIC S9(6)V99 COMP-3.

The types of numeric fields you will commonly see in COBOL layouts are:

* Display (including Signed fields)
* Binary
* Computational, or comp
* Comp-1
* Comp-2
* Comp-3

Display, including "Signed" or "Zoned" fields, is the most common, and comp-3 is the second most common type of numeric field.  Some compilers may also have comp-4 and comp-5 data types, usually to emulate comp on another compiler.

**Usage is Display**

**Display** format is the default for numbers in COBOL. If no "usage is" clause is specified, the default is "usage is display", which means the value is stored as EBCDIC characters (digits), as opposed to binary. The value may or may not have a decimal -- implied or real -- and may be unsigned or have an embedded or a separate sign -- which can be either leading or trailing.  The default "signed display" format field contains an embedded trailing sign, and is commonly called a "Signed", or "IBM Signed", or "Zoned" field.  This data type is described below.

**Signed Fields**

There is a common numeric data type used in COBOL on IBM mainframes called "**Signed**" (also called "**IBM Signed**", or "**Zoned**"). COBOL represents this type of field by an "S" in the picture clause of a display format field, e.g. PIC S9(6).  A Signed field is composed of regular EBCDIC numeric characters, one character per byte, for all digits except the one that holds the sign, either the most-significant (**sign leading**) or the least-significant (**sign trailing**) digit -- usually the least-significant digit. The digit that holds the sign combines, or "**over punches**" the sign of the number onto that digit.  This saves one byte that the sign would otherwise occupy. The value of that digit is stored as a binary value, and is OR'd with the sign code, which is D0 hex for negative numbers, C0 hex for positive values, and F0 hex for "unsigned" values.

Because of the overpunch, the digit that holds the sign will not appear as a number when the field is viewed in EBCDIC *character mode*.  If you have the field

   05  ACCOUNT-BALANCE         PIC S9(6)V99.

and view a value of 1.23 with an EBCDIC editor, it will read "0000012C".

ASCII COBOL compilers also use a Signed data type with an overpunch, but the sign bits are different and not standardized between compilers.  See our Tech-Talk brief [Signed Fields](http://www.3480-3590-data-conversion.com/article-signed-fields.html) for further details on both EBCDIC and ASCII Signed fields.

**Sign is Separate**

COBOL signed fields embed the sign in the value by default (see signed fields above).  But there is a provision in COBOL for a separate sign, and it can be either leading or trailing. The statement for this is

       SIGN IS SEPARATE  (or SIGN SEPARATE)

This may be combined with the leading or trailing clause:

       SIGN IS LEADING SEPARATE  (or SIGN LEADING SEPARATE)

   or  SIGN IS TRAILING SEPARATE

This statement can be applied to an elementary item (field) or to the entire record.

**Computational and Binary Fields**

Because computers perform computations with binary numbers, it is more efficient to store those values in the file in their native binary form than to store them in human readable base ten.  If the number is stored in its native binary format it can be input from the file and used directly.  If it's stored in a base ten format it needs to be converted to binary before performing calculations on it, then converted back to base ten for storage.

COBOL defines several binary data types.  We will list a brief summary here, and you can find more detail in [COBOL Computational Fields](http://www.3480-3590-data-conversion.com/article-cobol-comp.html) and in [COBOL Comp-3 Packed Fields](http://www.3480-3590-data-conversion.com/article-packed-fields.html).  Before we start, there is one important point to understand:  The COBOL standard leaves the actual implementation of most data types up to the vendor who wrote the COBOL compiler.  The reason for this is because different computers -- CPUs -- use different binary representations internally, and function best with their own type of binary numbers.  This approach results in better and faster compilers, but also causes confusion, because a "comp" data type on one machine is not necessarily the same as "comp" on another machine.  The table below lists the common uses; not all compilers will follow these types.  For more details on word order and signs see the link above.

Which data type a field uses for storage is determined by the "usage is" clause in the field definition, and in most cases the number of bytes of storage is determined by the number of digits in the PIC. Floating point numbers follow standard binary formats and as such their sizes are not determined by a PIC, and no PIC is used in the field definition. 

|  |  |
| --- | --- |
| **Data Type** | **Description of how this data type is stored** |
| Binary | This is a pure binary number, usually in 2's-complement, and usually either 2, 4, or 8 bytes. |
| Comp | The COBOL standard intends that the comp data type be implemented using the most efficient data type for a particular machine.  The compiler vendor will chose the best type for the CPU, probably binary. |
| Comp-1 | This is generally a single precision floating point data type. |
| Comp-2 | This is generally a double precision floating point data type. |
| Comp-3 | Comp-3 is very common and its format is nearly universal across platforms. It "packs" two digits into each byte. See [COBOL Comp-3 Packed Fields](http://www.3480-3590-data-conversion.com/article-packed-fields.html) for a full description of this data type. |
| Packed Decimal | Packed decimal is usually implemented as comp-3. See comp-3. |

When reading a binary or comp field specification, the size listed in the PIC is the number of decimal digits *after* the number is converted from binary to base ten.  In the case of a packed field, it's the size after unpacking. 

**Real Decimal**

Most PC programmers tend to think in terms of "real decimal" in numeric values.  On a PC, if you have a dollars and cents field for, say, invoice total, in the amount of $123.45, the file will contain the *six*bytes "123.45" (and probably a sign).  In other words, there is a *real decimal point*in the file.  COBOL can do this, too, via the following:

   05  INVOICE-TOTAL           PIC 999.99.

OR:

   05  INVOICE-TOTAL           PIC 9(3).9(2).

The presence of the "." in the PIC causes a real decimal in the file. Implied decimal, however, is much more common in COBOL.

**Implied Decimal**

Implied decimal simply means there is a decimal point *implied* at a specified location in a field, but not actually stored in the file. The location of the implied decimal is indicated by a "V" in the PIC. Using implied decimal saves space in the file. Implied decimal can apply to any kind of numeric field, including a packed, or comp-3 field.

For example,

   05  ACCOUNT-BALANCE     PIC 9(6)V99.

is an implied decimal field.  There are 6 digits, then an *implied* decimal - the V - and 2 more digits, for a total of 8 digits.  The field is 8 bytes in size; there is no "." in the file -- the location of the decimal point is *implied* to be between the 9(6) and the 99.  If the field contains "00000123" then the account balance is $1.23, because there is a decimal *implied* between the dollars and cents. 

**Synchronization and Alignment**

This topic is a bit involved for this tutorial, but you should be aware of it. When using binary storage (binary and comp), some compilers on some machines may require that a numeric field start on some boundary.  For example, on a 32 bit machine, it may require that a comp field start on a 32 bit boundary.

If you specify a comp field in the middle of a record, and it doesn't happen to begin on a 32 bit (4 byte) boundary, the compiler will "align" it to a 32 bit boundary to "synchronize" it.  What's actually stored in the file may not be the same as the PICs on the layout indicate.  This is not a very common problem, partly because binary and comp fields are not very common in files, but you should be aware of it.

**Part 5: Tables and Occurs**

A powerful feature of COBOL is the use of **tables**, via the "**OCCURS**" and "**OCCURS DEPENDING ON**" clauses.  This section describes COBOL Tables and the OCCURS and OCCURS DEPENDING ON clauses, both of which cause fields or groups to repeat some number of times.

**Contents of this section:**

|  |
| --- |
| **Problems with COBOL Occurs?**[Request a COBOL quote](http://www.3480-3590-data-conversion.com/quote2.html) **We know how to handle them!** |

   [Tables and the OCCURS clause](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-5.html#Occurs)   
   [Occurs Depending On](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-5.html#Occurs Depending On)

**Tables and the OCCURS clause**

Suppose you wanted to store your monthly sales figures for the year. You could define 12 fields, one for each month, like this:

   05  MONTHLY-SALES-1    PIC S9(5)V99.

   05  MONTHLY-SALES-2    PIC S9(5)V99.

   05  MONTHLY-SALES-3    PIC S9(5)V99.

   ...

   05  MONTHLY-SALES-11   PIC S9(5)V99.

   05  MONTHLY-SALES-12   PIC S9(5)V99.

But there's an easier way in COBOL.  You can specify the field *once* and declare that it repeats 12 times. You do this with the OCCURS clause, like this:

   05  MONTHLY-SALES  OCCURS 12 TIMES  PIC S9(5)V99.

(By now you should also know this can be written on two lines like this):

   05  MONTHLY-SALES  OCCURS 12 TIMES

                                   PIC S9(5)V99.

This specifies 12 fields, all of which have the same PIC, and is called a table (also called an array).  The individual fields are referenced in COBOL by using **subscripts**, such as "MONTHLY-SALES(1)".  This table occupies 84 bytes in the record (12 \* (5+2)). (The sign is embedded, not separate, and the decimal is implied.)

The OCCURS can also be at the group level, and this is the most useful application of OCCURS.  For example, all 25 line items on an invoice (75 fields) could be held in this group:

   05  LINE-ITEMS OCCURS 25 TIMES.

       10  QUANTITY            PIC 9999.

       10  DESCRIPTION         PIC X(30).

       10  UNIT-PRICE          PIC S9(5)V99.

Notice the OCCURS is listed at the group level, so the entire group occurs 25 times. The order of the data in the file is as-if you had specified multiple groups, like this:

   05  LINE-ITEMS-1.

       10  QUANTITY            PIC 9999.

       10  DESCRIPTION         PIC X(30).

       10  UNIT-PRICE          PIC S9(5)V99.

   05  LINE-ITEMS-2.

       10  QUANTITY            PIC 9999.

       10  DESCRIPTION         PIC X(30).

       10  UNIT-PRICE          PIC S9(5)V99.

       ...

   05  LINE-ITEMS-25.

       10  QUANTITY            PIC 9999.

       10  DESCRIPTION         PIC X(30).

       10  UNIT-PRICE          PIC S9(5)V99.

There can be **nested** occurs -- an occurs within an occurs.  In the next example, suppose we stock ten products and we want to keep a record of the monthly sales of each product for the past 12 months. We could do just that with this table:

   01  INVENTORY-RECORD.

       05  INVENTORY-ITEM OCCURS 10 TIMES.

           10  MONTHLY-SALES OCCURS 12 TIMES  PIC 999.

In this case, "INVENTORY-ITEM" is a group composed only of "MONTHLY-SALES", which occurs 12 times for each occurrence of an inventory item.  This gives an array (table) of 10 \* 12 fields.  The *only* information in this record are the 120 monthly sales figures -- 12 months for each of 10 items.

We could also have a description for each item. The description would go under the 05 level INVENTORY-ITEM group, at the 10 level, the same as the monthly sales.  Further, we could track, say, the sale price of each item for each month.  A record which will do these things is:

   01  INVENTORY-RECORD.

       05  INVENTORY-ITEM OCCURS 10 TIMES.

           10  ITEM-DESCRIPTION               PIC X(30).

           10  MONTHLY-SALES OCCURS 12 TIMES.

               15  QUANTITY-SOLD              PIC 999.

               15  UNIT-PRICE                 PIC 9(5)V99.

Notice we have made MONTHLY-SALES a group, which now contains two fields, and the whole group repeats 12 times for each instance of INVENTORY-ITEM.  This short layout has 250 fields: two fields (QUANTITY-SOLD and UNIT-PRICE) that repeat 12 times for each inventory item, times 10 items, plus the ITEM-DESCRIPTION field for each of the 10 items.  Fields and groups can be nested several levels deep, and it's possible to have thousands of fields in a layout only a couple pages long. 

**Occurs Depending On**

One really great feature of COBOL tables, and a really nasty one to convert to other languages, is the "OCCURS DEPENDING ON".  This is an OCCURS, like above, but the number of times it occurs *in a particular record* can vary (between some limits). The number of times it actually occurs in any particular record will be given by a value in another field of that record. This creates records that vary in size from record to record.

The OCCURS-DEPENDING-ON can include many subordinate fields and groups, all of which occur multiple times.  Further, most compilers allow one or more (fixed) OCCURS to be nested within an OCCURS-DEPENDING-ON, and some compilers allow multiple OCCURS-DEPENDING-ON to be nested, or to occur in succession.  This can get pretty involved, so we will only give one simple example, that of a patient's medical treatment-history record .

   01  PATIENT-TREATMENTS.

       05  PATIENT-NAME                PIC X(30).

       05  PATIENT-SS-NUMBER           PIC 9(9).

       05  NUMBER-OF-TREATMENTS        PIC 99 COMP-3.

       05  TREATMENT-HISTORY OCCURS 0 TO 50 TIMES

              DEPENDING ON NUMBER-OF-TREATMENTS

              INDEXED BY TREATMENT-POINTER.

           10  TREATMENT-DATE.

               15  TREATMENT-DAY        PIC 99.

               15  TREATMENT-MONTH      PIC 99.

               15  TREATMENT-YEAR       PIC 9(4).

           10  TREATING-PHYSICIAN       PIC X(30).

           10  TREATMENT-CODE           PIC 99.

Here are the significant points of this record:

1. The name of the record is "PATIENT-TREATMENTS".
2. The first three fields "PATIENT-NAME", "PATIENT-SS-NUMBER", and "NUMBER-OF-TREATMENTS" occur in the **fixed portion** of every record.  This fixed portion is the same for every record.
3. The TREATMENT-HISTORY group is the **variable portion** of the record. It can occur from 0 to 50 times.
4. ""NUMBER-OF-TREATMENTS" is a number from 0 to 50 that tells us how many times the group TREATMENT-HISTORY occurs in *this* record.
5. The value in NUMBER-OF-TREATMENTS is stored in a comp-3 packed format. This is very common. Also very common is comp or binary format.  All of these are binary data formats.
6. TREATMENT-HISTORY is a group that is comprised of all the lower level fields beneath it. (Down to the next 05 level, or the end of the record).
7. All the fields and groups within TREATMENT-HISTORY occur between 0 and 50 times.
8. Because 0 is a valid number of occurrences, it is possible the variable portion of the record is not present.
9. The "INDEXED BY TREATMENT-POINTER" clause may or may not be present.  If present it tells the compiler the name of the variable (TREATMENT-POINTER) to use as the index into the array.  If you don't understand this, you can safely ignore the "indexed by..." clause, unless you are programming in COBOL.
10. TREATMENT-DATE is a group that is comprised of the day, month, and year fields beneath it.
11. These records vary in size from 41 to 2041 bytes, and would be stored in some type of variable length file.

Our article [Occurs...Depending On](http://www.3480-3590-data-conversion.com/article-cobol-occurs.html) discusses this further, and has suggestions for converting variable tables to PC files.

## Part 6: Redefined Records

**Contents of this section:**

|  |
| --- |
| **Problems with Redefined Records?**[Request a COBOL quote](http://www.3480-3590-data-conversion.com/quote2.html) **We know how to handle them!** |

   [Examples of redefined records](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-6.html#Examples)   
   [Converting to a PC database](http://www.3480-3590-data-conversion.com/article-reading-cobol-layouts-6.html#Converting to PC)

**Examples of redefined records**

Complex data sets usually cannot store all their data in just one record type, so they have multiple record types.  For example, medical records may have a record type to identify a patient (name, address, etc.), another record type for treatment data, and a third for payment information. (Let's call them A, B, and C records).  When all these record types are stored in one file, that file has "multiple record types", or "redefined records".

When a file contains multiple record types (redefined records), it often associates these records in groups.  Using the medical file example above, one patient's data might consist of a patient ID (A) record followed by one treatment (B) record followed by a payment (C) record.  Another patient's data might consist of a patient ID (A) record followed by 5 treatment (B) records, followed by 3 payment (C) records.  These patient records are often associated via a key field, but some times the order in the file is the only thing that ties them together.

Another common file structure containing multiple record types is a file that starts with a header (H) record, say to identify the vendor and run date, followed by the data records, and ending with a trailer (T) record which contains a record count and totals of the individual records. These totals are often used as a check of the data after importing; the sum of the individual records should equal the total in the trailer record.

A typical COBOL medical file might contain both these concepts. It may start with a header (H) record, followed by groups of A, B, C records (one group for each patient), and end with a trailer (T) record. That gives you a file with 5 record types.

**Converting to a PC database**

Most PC databases can make use of relational tables (or files), but usually can't deal with all these record types in one file.  There are several ways to deal with this, and the best choice will be dictated by both the COBOL file structure and the PC application requirements. Some of the solutions are:

**1. Remove record types you don't need.**  
Sometimes only one of the record types is needed. For example, a file may contain a header record that identifies the vendor or receiver, then detail records (the data you actually need), followed by a trailer summary record. You can discard the header record, keep the data records, and discard the trailer record (you might want to check the header and trailer records first!). Now you have only one record type.

**2. Simply write each record type to a separate file.**  
Occasionally you can separate each record type into a separate file without relating them. This is seldom the case, but when it is an option, it's usually the easiest way to go.  Using the medical example above, if you wanted to analyze the geographic distribution of patients who checked into the hospital, you would only need record type A. And if you wanted to *separately* analyze the treatments they received, you would only need the B records. You would not need an association between the A and B records for either analysis.

**3. Append multiple record types into one new type.**  
This works best if there is a one-for-one correspondence of a small number of record types. For example, a single A record followed by a single B record. You can append record B to record A, creating a new record, A+B.  If you have one A record followed by several B records, it can get messy. It's generally best to relate these, but you might be able to prepend the A record to each B record, depending on the data and the application. You need to be careful, as that will multiply the number of A records. If you later forget you've done this and, say, sum the values in all the A records, you will get an inflated value.

**4. Combine all the needed data into one new record.**  
When there are too many records for method 3 to be effective, you might write a COBOL program to read all the related records into memory, then write a completely new output record. The new record could be whatever you wish, mixing data from the multiple records read, perhaps even summing the detail records. This is particularly useful if you are analyzing complex data, and all you need are the totals, not all the detail. For example, you might sum all the weekly sales records for a product and generate one record with the annual total. That would compress 53 records (1 A record + 52 B records) into one new record.

**5. Split the data into multiple files, one for each record type.**  
Then you can import them into separate tables in a PC database and relate them. This is the method usually used, and the "proper" way to convert a multiple record COBOL file to a relational PC database. The original records often have a key that is common to all record types, which can be used to relate the PC tables. If there is no key, DISC can generate one when we do the conversion. Generating a key, however, requires understanding the data sufficiently well to properly group the related records. So it may not be a trivial task.

There are variations on these themes. For example, in a file with 6 record types, you might start by discarding the header and trailer, then combine two records into one, then add a key and write the three resulting record types to three files, for import into three tables in the PC database.

Making the best choice requires knowledge of the source data, the PC application or language requirements, and experience with manipulating such data.

## Part 7: Conclusion

|  |
| --- |
| **Need to convert COBOL files?**[Request a COBOL quote](http://www.3480-3590-data-conversion.com/quote2.html) **That's our business!** |

COBOL is a capable language for business and financial applications.  It was designed to accommodate complex data sets while efficiently using computer resources, especially disk space.  The ability to handle multiple record types in a file, to re-use portions of the record (redefined fields), and to permit repeating groups of fields (tables) gives it advantages that other languages didn't have for years after COBOL's introduction.

But it's those very features that can make it difficult to convert COBOL files to PC files.  Most PC applications do not understand any of these concepts, and trying to deal with a complex COBOL file on a PC can be a nightmare.  Disc Interchange deals with these situations every day, and we can offer you several options to cope with these issues when converting a mainframe COBOL file to a PC database.  We often try to simplify the file, split it into multiple PC files, one for each record type, and convert the OCCURS-DEPENDING-ON to multiple PC records. Or if necessary, we can convert the OCCURS-DEPENDING-ON to a fixed number of occurrences.  And of course we always convert comp, comp-3, and Signed fields to leading-sign ASCII numeric fields for the PC.

Although this article presented most common COBOL file features, there are a number of variations that we haven't covered in this tutorial.  But if you understand all the basic examples above, you can manage to read most simple layouts by combining these rules.  Be aware, however, that some data types are dependent upon the COBOL compiler and CPU it runs on (see part 4, Numeric Fields, and our article [COBOL Computational Fields](http://www.3480-3590-data-conversion.com/article-cobol-comp.html)).

If you combine all the concepts above and expand the size to several hundred fields, you will get a typical COBOL file such as a medical or financial file. We've seen layouts of 100 pages that contained thousands of fields in a dozen record types.  Often there's a mix of binary and character fields redefining each other, within multiple OCCURS.  Devising a way to get such data into a PC database can be challenging, but it is possible.

COBOL layouts are one of the best known and understood methods of specifying a record layout.  If you process data from a mainframe, you will certainly encounter COBOL layouts.  Investing a little time to understand how to read the layouts, and becoming aware of pitfalls like redefined fields and records, could save you significant grief later.  With a little practice you should find COBOL layouts easy to read, efficient, and concise.

#### Author's notes:

**Length of article:**You are likely to encounter each of the above concepts, so I felt it was important to present all of them, and give brief examples.  But I didn't want this to be overwhelmingly long, so I opted for brevity.  However, I feel this tutorial may be *too* brief to adequately describe some issues.  I welcome feedback on anything that was incomplete, unclear, or omitted.  If you wish to comment, please send an email to [TechTalkEditor@discinterchange.com](mailto:TechTalkEditor05@discinterchange.com)   Please put "COBOL" in the subject line.

**Questions about COBOL:**While DISC remains committed to providing personalized help to our conversion customers, regrettably I am no longer able to respond to general questions about how to write COBOL programs.  The volume of questions is overwhelming for the time I have available, and it was never my intent to teach COBOL programming.  There are other web sites and many good books written to address COBOL programming.   I sincerely hope this article has been of help to you.

# COBOL Comp-3 Packed fields:

|  |
| --- |
| **Need to convert Packed fields?**[Request a COBOL quote](http://www.3480-3590-data-conversion.com/quote2.html) **That's our business!** |

This page discusses how data is stored in COBOL "comp-3", or "packed" fields. 

## The COBOL "Comp-3" Data Type (also called "Computational-3", "Packed Decimal", or "Packed")

(See note 1 about terminology)

COBOL Comp-3 is a binary field type that puts ("packs") two digits into each byte, using a notation called Binary Coded Decimal, or BCD.   This halves the storage requirements compared to a character, or COBOL "display", field.  Comp-3 is a common data type, even outside of COBOL, and is fairly standard across platforms -- that is, it is not dependent upon the operating system, language, or CPU, as the COBOL "comp" is. (See [COBOL Computational Fields](http://www.3480-3590-data-conversion.com/article-cobol-comp.html) for information on the comp data type).  However, comp-3 is not commonly found in PC languages.

## How the data is stored

The Binary Coded Decimal (BCD) data type is just as its name suggests -- it is a value stored in decimal (base ten) notation, and each digit is binary coded. Since a digit only has ten possible values (0-9), it can be represented in binary form with only 4 bits. Four bits is called a "nybble", and each nybble contains one digit of the value.  Therefore, you can get two digits in each 8 bit byte. (There's an example below).  Normal EBCDIC or ASCII character representation (COBOL "Display") only stores one character (digit) per byte, so packed data only requires half the storage of unpacked (character) data.  (See [Character, BCD, and Binary Fields](http://www.3480-3590-data-conversion.com/article-bcd-binary.html) if this description is not clear.)

The value in a comp-3 field is stored high-order to low-order.  That is, the upper nybble of the first byte encountered in the file is the most significant digit of the value, the lower nybble of that byte is the next digit, etc., etc.  The last nybble -- the low nybble of the least significant byte -- is used to store the sign for the number. Unlike IBM Signed fields (see [EBCDIC to ASCII Conversion of Signed Fields](http://www.3480-3590-data-conversion.com/article-signed-fields.html)), this nybble stores *only*the sign, not a digit.  "C" hex is positive, "D" hex is negative, and "F" hex is unsigned.  In COBOL comp-3 fields (and in most other languages) this nybble is reserved for the sign whether or not the field is denoted as a signed field in the COBOL PIC.

Comp-3 packed fields are aligned on byte boundaries, and the field is always a whole number of bytes.  The sign nybble is always the low nybble of the LSD (least significant digit).  Since the sign takes one nybble, and because there are always an even number of nybbles in any number of bytes, an *odd* number of digits will fully-fill a comp-3 field. (An *odd* number of digits plus a sign nybble makes an even number of nybbles, or fully-filled bytes).  If the size of the field is specified as an even number of digits, as in "PIC S9(6) comp-3.", the upper nybble is ignored and is usually, but not always, set to zero.

## Comp-3 fields in COBOL

Comp-3 fields are denoted in COBOL with the "usage is" clause after the PIC, like this:

PIC S9(5) usage is computational-3.

However, the "usage is" is not required and seldom used, and "computational-3" is usually abbreviated "comp-3", so you more commonly see:

PIC S9(5) comp-3.

The COBOL PIC, or picture, for a comp-3 packed field specifies the number of digits *after unpacking*.  The actual number of bytes occupied in the file is about half that.  To calculate the number of bytes from the PIC, add 1 (for the sign) to the total number of digits, divide by 2, and round up if necessary.  For example:

PIC S9(7) COMP-3. Byte size = (7 + 1) / 2 = 4

PIC S9(5)V99 COMP-3. Byte size = (5 + 2 + 1) / 2 = 4

PIC S9(6) COMP-3. Byte size = (6 + 1) / 2 = 3.5, rounded to 4

Comp-3 fields reserve a nybble for the sign, even for "unsigned" values, so the following fields are *still* 4 bytes:

PIC 9(7) COMP-3. Byte size = (7 + 1) / 2 = 4

PIC 9(6) COMP-3. Byte size = (6 + 1) / 2 = 3.5, rounded to 4

## Examples

Lets look at some examples of how comp-3 data is stored. The left column in the table below is the decimal value being stored, and the right column is the hexadecimal value you will see in the file:

Value Comp-3, hex

+0 0C

+1 1C

+12 01 2C

+123 12 3C

+1234 01 23 4C

-1 1D

-1234 01 23 4D

Each underlined value above represents one byte, in hexadecimal (hex) form.  We have only used as many bytes as needed to store the value shown on the left.

When you "unpack" a packed, or comp-3, field, the size of the field will double.  This will cause all fields following it to shift down.  If the field is in a redefined area, it will likely no longer fit in the allocated space, and the original field it redefined will have to be modified, or filler will have to be added, to accommodate the larger unpacked field. Just a few comp-3 fields can make for a messy situation, affecting many other fields, and even other records if the file contains multiple record types.