PRECISION HEALTHCARE SOLUTIONS

Mastering Discern Explorer

John Simpson

Introduction

Discern Explorer (CCL) has a poor reputation in the industry. Many of my clients believe that CCL reports are inefficient and difficult to maintain. For an exceptionally large number of hospitals, this statement is unfortunately very true.

The good news is that this doesn’t have to be the case. CCL reports can be very easy to write and maintain. Additionally they can be written so that they perform exceptionally well. By using the techniques taught in this manual, you will learn how to create fast performing reports that are simple to code and maintain.

The format of this book assumes you have a basic knowledge of CCL and know how to create a SELECT statement and assign variables. Instead of teaching the basics, this book will give you insight into some advanced CCL programming and techniques for simplifying your code. Additionally, where possible I will give pointers on things not to do in your reports.

Now, let’s move on to the good stuff.

Contents

Programming Practices

Implementing Standards

The absolute first step in establishing your CCL shop is to develop and implement coding standards. Your standards should include policies and procedures that cover at least the following topics:

- Source code naming standards
- Location of source code on the Cerner server
- Standard source code headings with detailed descriptions of the purpose of the script along with any special instructions that are needed (e.g. servers needing cycling, code value setup, etc.)
- Code testing, promotion and synchronization procedures
- Documentation standards.

By implementing and committing to strong standards you help ensure that your entire development team is working in a consistent manner which helps reduce future code maintenance time.

While the development of standards is outside the scope of this manual, it is a very important topic that should be covered in your organization. If you require assistance in developing your CCL report writing standards, Precision Healthcare Solutions can be contracted to develop your standards with your team.

Simplifying Your CCL Scripts

I mentioned in the introduction that your CCL reports can be easy to write and maintain. This is primarily accomplished by simplifying your CCL scripts.

The standard practice in any programming language is to teach a student how to use as many functions and tools as possible in the allotted time. This is true for common languages such as Java and Visual Basic and is also just as true for Discern Explorer.

Unfortunately with CCL, this training practice puts techniques in the hands of programmers that should rarely be used (if ever). These techniques appear to be exactly what a programmer needs for many programming tasks and are what are commonly used which result in inefficient and difficult to maintain CCL code.

The traditional CCL training model teaches students how to use a wide variety of JOIN types to create a single SQL query capable of retrieving all or most of the data needed for a report. Unfortunately, by using this method, a programmer must modify and re-test the entire query to ensure its accuracy once a change has been made.

Throughout this document I'll be detailing techniques you can use to break a large JOIN into multiple SELECT statements which can be used to populate a record structure and finally output your report. By using these techniques, you will be able to add and modify functionality in a CCL report with little impact on the code that existed previously.

By using the techniques described in this book, there is absolutely no reason you will ever need to use poor performing joins such as DONTCARE and ORJOIN.

In addition to breaking your code into smaller manageable pieces, it is also important that you learn to eliminate extra code that serves no purpose. For example, I see many programs where variables are assigned a temporary value on one line and immediately assigned the correct value on the next line.

e.g.

```
SET cvAUTH = 0
SET cvAUTH = UAR_GET_CODE_BY(“MEANING”, 8 ,”AUTH”)
```

While this works and may not seem like a big issue, it just leads to a script with more lines of code than necessary. By removing the first line, the script will run exactly the same and have one less line of code to maintain.

You should also take as many steps as possible to eliminate extra code from your SELECT statements. For example, I have seen many scripts that check to see if the BEG_EFFECTIVE_DT_TM is less than the current date time. Never in my 10 years of CCL development have I ever seen a single case where the BEG_EFFECTIVE_DT_TM is greater than the current date and time. This date is primarily used as a date/time stamp of when the record was created.

You can safely remove all checks for BEG_EFFECTIVE_DT_TM being less than the current date and time from your code.

By eliminating code that is not needed, you are making future maintenance of a program easier. If another programmer or you have to modify a script in the future, the fewer lines of code to examine the less time will be needed to make the modifications.

The next step in simplifying your code requires a solid understanding of record structures.

Test Your Code in the Back-End

One of the problems I have seen frustrating programmers for the past few years is how little error reporting Discern Visual Developer provides. Programmers write their scripts and click the execute button only to receive a cryptic error message back. Sometimes this is enough to find the bug and fix it, but more often than not, if the programmer had simply run their CCL script from the back-end on the CCL command line, they would have seen exactly where in the execution of the script the errors occurred.

Record Structures

Overview of a Record Structure

A record structure is an area defined in memory that provides a place to store information. You can store any type of information in a record structure including both calculated values and data read from the Cerner database.

Record structures can be treated like tables and used to join to other record structures as well as tables in the Cerner database.

When used correctly, a record structure can help you simplify a complex SQL statement and create very fast efficient reports.

Defining a Record Structure

To use a record structure, you must first define it. This is done with the RECORD command followed by the name of your structure.

e.g.

```
RECORD MyRecord
```

The next step in creating your structure is to define the fields that will exist in your structure. To do this, you need to identify a depth, field name, field type and field size.

e.g.

```
RECORD MyRecord (
     1 NameLast                  = c20
     1 NameFirst                 = c15
     1 AgeInYears                = i4
)
```

Let's explain the example above in some greater detail. First, you may have noticed that a left parenthesis has been added after the record structure name and a right parenthesis has been added at the end of the definition. Your field definitions must always be within a set of opening and closing parenthesis.

Next, I have defined all of these elements at a depth of one. Your top-most level must always start at one. I will explain the purpose of multiple depth record structures in a moment.

After the depth, I define the name of the field. The name can be anything you want except that it must be unique within the current depth of the record structure. So, you cannot have two fields at level one called NameLast.

Finally, I define the field type followed by size of field. Common field types available are:

c – character fields store textual data and can be virtually any size you desire. The larger the field however the greater amount of memory you will use.

i – integer fields are used to store whole numbers (negative or positive). Typically you will use i4 when defining integer fields.

f – floating point variables are numbers that can store decimal points. I typically define all of my floating point fields as f8.

dq8 – date/time fields are used to store the numerical representation of an Oracle date/time.

To store data in the record structure defined above, we would access the field names directly in the following manner.

```
MyRecord->NameLast = “Simpson”
MyRecord->NameFirst = “John”
MyRecord->AgeInYears = 36
```

To read the record structure fields, we would use the same field names.

```
COL 0, MyRecord->NameLast
COL 32, MyRecord->NameFirst
COL 50, MyRecord->AgeInYears
```

Multiple Depth Record Structures

In the previous topic, I described how to create a simple record structure. While a simple single level record structure does have some applications, it is not where record structures offer their greatest power.

In all complex CCL reports, we are expected to pull information from a variety of sources in the Cerner database. It is not uncommon to have a single report that needs information from registration, labs, pharmacy, radiology and other modules.

If you were to attempt this task with a single SQL statement; you would find yourself caught with a very large SELECT statement littered with OUTERJOINS and DONTCARE joins. While this works, some problems arise:

1. The SELECT statement may be inefficient and cause Oracle to work harder than it has to. This results in slower performing reports.
2. Your code becomes very difficult to follow and it becomes increasingly easier to make mistakes.
3. This type of SELECT statement often brings back more rows than you want to work with. For example, if you have a patient that returns 10 rows of labs and 5 rows of radiology results, you would see rows of data returned. Managing this volume of data in your report would quickly become tedious.

The ideal situation would be to break this large SQL statement into smaller more manageable parts. This is where the use of record structures comes in. With a record structure in place, you could start with a SQL statement that reads your registration information and initially populates the record structure. The next step would read the lab results and populate another area of the structure, followed by pharmacy, radiology and any other module.

The benefits of this technique are numerous. Some of these benefits are:

1. Your code is easier to read and maintain in the future. If you need to make a change to the lab section, you can do so without breaking the rest of the program.
2. Identifying bugs also becomes easier as you only have to diagnose a small area of code rather than a larger SQL statement.
3. Oracle doesn't have to work so hard to retrieve your data and reports run faster. You can also manipulate your queries so they make better use of available indexes. I personally have seen clients receive a boost of 400% and higher using this method.
4. You can have better control of sorting your data. You may want to sort a report by the registration data (e.g. Nurse Unit followed by Patient Name), but how do you want the lab, radiology and pharmacy data sorted? With a single select, your control is limited. Using a record structure, you can sort your data as it is loaded into the structure and print it in sorted order later.

Multiple Depth Record Structures Example

Explaining how to use a multiple depth record structure is best done in an example format. I will start off simple and move into more advanced topics such as adding a third level of depth.

In this example, we are going to start by collecting current census along with patient specific information such as MRN and FIN #. Later on, we will gather a list of Orders placed with the past 24 hours.

First, I like to collect the code values for our SELECT statement and assign them to variables. I do this instead of hard coding the code value because using a variable name is programmer friendly and you avoid the risk of the actual code value being different between domains. *(Please note that the Display Key Values shown in the example below may be different at your site.)*

```
SET cvMRN = UAR_GET_CODE_BY("DISPLAYKEY",319,  "MRN")
SET cvFIN_NBR = UAR_GET_CODE_BY(“DISPLAYKEY”,319,”FIN NBR”)
SET cvINPATIENT = UAR_GET_CODE_BY(“DISPLAYKEY”,71,”INPATIENT”)
```

Our next step is to define our record structure.

```
RECORD rDATA (
     1 COUNT                    = i4
     1 DATA[*]
          2 ENCNTR_ID           = f8
          2 NURSE_UNIT          = c20
          2 ROOM                = c10
          2 BED                 = c5
          2 PATIENT_NAME        = c35
          2 MRN                 = c10
          2 FIN_NBR             = c10
          2 ORDER_COUNT         = i4
          2 ORDERS[*]
               3 ORDER_DT_TM    = c16
               3 ORDERABLE      = c40
)
```

This structure is a little different from the one we defined in our first example. First, we are going three levels deep and we have also created two fields without a type and added square brackets and an asterisk next to the field name.

The fields with the square brackets and asterisks are used to identify these fields as being able to support more than one set of values. This means that if we want to take all of the records returned from a SQL statement and put them into the same structure, we can do so. The asterisk means that the structure has an undefined size that can be altered at runtime.

If desired, you could actually hardcode a number into the square brackets. This would fix the number of records to that size. An example of why you would do this may be “Show me the last 10 lab orders placed on a patient”.

Next, we are going to populate our record structure with our base census.

```
SELECT INTO “NL:”
      ENCNTR_ID                 = E.ENCNTR_ID,
      NURSE_UNIT                = UAR_GET_CODE_DISPLAY(E.LOC_NURSE_UNIT_CD),
      ROOM                      = UAR_GET_CODE_DISPLAY(E.LOC_ROOM_CD),
      BED                       = UAR_GET_CODE_DISPLAY(E.LOC_BED_CD),
      PATIENT_NAME              = P.NAME_FULL_FORMATTED,
FROM  ENCNTR_DOMAIN             ED,
      ENCOUNTER                 E,
      PERSON                    P
PLAN  ED
      WHERE ED.END_EFFECTIVE_DT_TM > CNVTDATETIME(CURDATE, CURTIME3)
JOIN  E
      WHERE E.ENCNTR_ID = ED.ENCNTR_ID
      AND E.ENCNTR_TYPE_CD = cvINPATIENT
      AND E.REG_DT_TM+0  > CNVTDATETIME(CURDATE, CURTIME3)
      AND E.DISCH_DT_TM = NULL
      AND E.ACTIVE_IND +0= 1
JOIN  P
      WHERE P.PERSON_ID = E.PERSON_ID
HEAD REPORT
      nCOUNT = 0
DETAIL
      nCOUNT = nCOUNT + 1
      STAT = ALTERLIST(rDATA->DATA, nCOUNT)
      rDATA->DATA[nCOUNT].ENCNTR_ID = ENCNTR_ID
      rDATA->DATA[nCOUNT].NURSE_UNIT = NURSE_UNIT
      rDATA->DATA[nCOUNT].ROOM = ROOM
      rDATA->DATA[nCOUNT].BED = BED
      rDATA->DATA[nCOUNT].PATIENT_NAME = PATIENT_NAME
      rDATA->DATA[nCOUNT].MRN = “ “
      rDATA->DATA[nCOUNT].FIN_NBR = “ “
FOOT REPORT
      rDATA->COUNT = nCOUNT
WITH NOCOUNTER
```

The first thing you will notice in the SQL statement is that I have made no OUTERJOINS or DONTCARE joins. Where possible, avoiding these types of joins improve efficiency of your report. Having one OUTERJOIN is fine (and recommended to save time), but more than one will lead to inefficiencies. DONTCARE joins should always be avoided as they are difficult to debug and very inefficient.

The SQL statement above only collects encounters and person details. The ENCNTR_ALIAS (FIN#, MRN) is not collected in the primary SQL statement. Instead, we collect our base information first, and worry about the other details later.

In the HEAD REPORT section, I initialize a counter variable to a value of zero. This counter will be used throughout the rest of the report writer section of the SQL statement to track the current position in our record structure.

The first line in the DETAIL section increments the counter variable by one. For every record returned in our SQL statement, the lines of code in the detail section will be executed again. So, for the second record returned in the SQL statement, the counter variable will be incremented by one again and will hold a value of two.

After incrementing our counter, we need to size our record structure so it can accommodate the number of records we have. Unfortunately, we can't determine the number of records that have returned in the SQL statement until all the records have been read. This of course makes it impossible to go back and use the values from each of those records.

To solve this problem, we need to grow the size of the record structure every time we read a record. This is done with the ALTERLIST function.

```
STAT = ALTERLIST(rDATA->DATA, nCOUNT)
```

In the example above, the size of rDATA->DATA will grow by the size of nCOUNT every time it is executed. The first time the DETAIL section in our report writer is run, the size will be one, and the second time will be two, and so on. While resizing, data that exists in the structure will remain intact.

After sizing our structure, we need to put some values in the record structure fields. This is done by simply assigning the value to the record structure field name. In the square brackets, we use our counter variable to indicate the position we would like to assign the value to.

```
rDATA->DATA[nCOUNT].ENCNTR_ID = ENCNTR_ID
rDATA->DATA[nCOUNT].NURSE_UNIT = NURSE_UNIT
rDATA->DATA[nCOUNT].ROOM = ROOM
rDATA->DATA[nCOUNT].BED = BED
rDATA->DATA[nCOUNT].PATIENT_NAME = PATIENT_NAME
rDATA->DATA[nCOUNT].MRN = " "
rDATA->DATA[nCOUNT].FIN_NBR = " "
```

You may have noticed in our code example above that two of the fields have been assigned an empty string as a value. These fields are the fields we have not collected yet. By default, a string field in a record structure is assigned a value of NULL. If for some reason, your code does not ever assign a value to a string field in your structure, it will cause abnormal printing issues. To remedy this problem, I assign a single space to these fields.

In the example above, a situation may arise where a MRN or FIN # record is not present. If this were to happen, the report would print incorrectly.

The final step in our first SQL statement is to record our counter to a variable we can use elsewhere. For simplicity, I keep the variable as a top level variable in my record structure.

```
FOOT REPORT
      rDATA->COUNT = nCOUNT
```

After collecting our primary data population, it is time to collect the other information we are interested in. Next, I will be collecting information from the ENCNTR_ALIAS table to record the MRN and FIN #. While these could have been added to the primary SQL statement, I would have had to join to the ENCNTR_ALIAS table two times. By using a record structure, I will reduce the load and only access the table once.

```
SELECT INTO “NL:”
      ALIAS       = EA.ALIAS
FROM  (DUMMYT                 D WITH SEQ=VALUE(rDATA->COUNT)),
      ENCNTR_ALIAS            EA
PLAN  D
JOIN  EA
      WHERE EA.ENCNTR_ID = rDATA->DATA[D.SEQ].ENCNTR_ID
      AND EA.ENCNTR_ALIAS_TYPE_CD IN (cvMRN, cvFIN_NBR)
      AND EA.ACTIVE_IND = 1
      AND EA.END_EFFECTIVE_DT_TM > CNVTDATETIME(CURDATE, CURTIME3)
DETAIL
      IF (EA.ENCNTR_ALIAS_TYPE_CD = cvMRN)
            rDATA->DATA[D.SEQ].MRN = ALIAS
      ELSE
            rDATA->DATA[D.SEQ].FIN_NBR = ALIAS
      ENDIF
WITH NOCOUNTER
```

The first thing that may be new to you is the use of the DUMMYT table. The DUMMYT table is a table in Cerner that has a single record and serves the purpose of being a placeholder. If you have used OUTERJOIN or DONTCARE joins you would have used the DUMMYT table before.

However, the use for record structures is a little different. The DUMMYT table is used to identify and represent the number of rows you have stored in your record structure. So, if you’re original SQL statement returns 85 patients in the census, the value of rDATA->COUNT will be 85 and 85 records will be returned on the DUMMYT selection.

This is accomplished with the following line:

```
FROM  (DUMMYT                 D WITH SEQ=VALUE(rDATA->COUNT))
```

The SEQ statement tells CCL to produce rDATA->COUNT worth of records when reading the DUMMYT table. In this example, we are assigning the DUMMTY table an ALIAS of D. Like all other tables, the alias naming is entirely up to you.

Please note that the brackets around the DUMMYT line are necessary and the code will not work without it.

Once we have generated the appropriate number of DUMMYT records, it is time to use those records to join to our ENCNTR_ALIAS table.

```
PLAN  D
JOIN  EA
      WHERE EA.ENCNTR_ID = rDATA->DATA[D.SEQ].ENCNTR_ID
```

In the example above, we are treating our record structure like an Oracle table. We may have 85 records in our record structure with DATA[] elements numbered 1 to 85. When we defined our DUMMYT table above, it made a field called SEQ with values 1 through 85. We use that field to JOIN to our record structure.

So, to Oracle, repeated iterations of the above WHERE statement would look something like:

WHERE EA.ENCNTR_ID = rDATA->DATA[1].ENCNTR_ID
WHERE EA.ENCNTR_ID = rDATA->DATA[2].ENCNTR_ID
WHERE EA.ENCNTR_ID = rDATA->DATA[3].ENCNTR_ID
etc.

Since each of the rows in our record structure have a valid ENCNTR_ID, the join to the ENCNTR_ALIAS table is a success and the rows are returned.

To save joining to the ENCNTR_ALIAS table twice, I've added a line to filter on the two ENCNTR_ALIAS_TYPE_CD fields we are interested in:

```
AND EA.ENCNTR_ALIAS_TYPE_CD IN (cvMRN, cvFIN_NBR)
```

This line will return both MRN and FIN # records. Later, in the DETAIL section of the code, I have an IF statement that assigns the MRN and FIN # values back to the record structure.

```
      IF (EA.ENCNTR_ALIAS_TYPE_CD = cvMRN)
            rDATA->DATA[D.SEQ].MRN = ALIAS
      ELSE
            rDATA->DATA[D.SEQ].FIN_NBR = ALIAS
      ENDIF
```

The wonderful thing about using record structures and multiple SQL statements is that if you have a coding error in one of your SQL statements, it will be much easier to identify where you need to make the changes. For example, if the code for collecting the MRN/FIN # fails, the rest of the code will still work correctly.

When you go to print your report, if the MRN/FIN # does not show up on the report, you will know exactly where to look to debug and make corrections.

Debugging with ECHORECORD

Cerner has a very handy command that can greatly help in debugging reports with record structures. The output of this command can only be seen on a back-end session but can save hours of debugging work.

The command is called ECHORECORD and is used to show the content of a record structure. For example, to print out the contents of our rDATA structure, add the following line to your CCL script and run the code:

```
CALL ECHORECORD(rDATA)
```

Next, we are going to complete our example from above and add the Orders information. To keep the example as simple as possible in order to focus on learning record structures, we are simply going to store any order placed in the last 24 hours. This could be Lab, Diets, Pharmacy or any other type of order. To show how the data can be sorted for report output, I'll sort this example in descending order by the original order date/time field.

```
SELECT INTO “NL:”
      ENCNTR_ID         = O.ENCNTR_ID,
      ORIG_ORDER_DT_TM  = FORMAT(O.ORIG_ORDER_DT_TM,”YYYY-MM-DD HH:MM;;Q”),
      ORDERABLE         = UAR_GET_CODE_DISPLAY(O.CATALOG_CD)
FROM (DUMMYT            D WITH SEQ=VALUE(rDATA->COUNT)),
      ORDERS            O
PLAN  D
JOIN  O
      WHERE O.ENCNTR_ID = rDATA->DATA[D.SEQ].ENCNTR_ID
      AND O.ORIG_ORDER_DT_TM >= CNVTDATETIME(CURDATE-1,CURTIME3)
ORDER ENCNTR_ID, ORIG_ORDER_DT_TM DESC
HEAD ENCNTR_ID
      nCOUNT = 0
DETAIL
      nCOUNT = nCOUNT + 1
      STAT = ALTERLIST(rDATA->DATA[D.SEQ].ORDERS, nCOUNT)
      rDATA->DATA[D.SEQ].ORDERS[nCOUNT].ORDER_DT_TM = ORIG_ORDER_DT_TM
      rDATA->DATA[D.SEQ].ORDERS[nCOUNT].ORDERABLE = ORDERABLE
FOOT ENCNTR_ID
      rDATA->DATA[D.SEQ].ORDER_COUNT = nCOUNT
WITH NOCOUNTER
```

As you can see in the code example above, the base SQL statement for collecting the orders is no different than collecting our MRN and FIN #. However, where the differences begin is with the ORDER command and the report writer section.

Since the orders are three levels deep in our record structure, we need to ensure that we don't lose track of our position. To do this, we must initially sort by a unique identifier in our structure. For this example, ENCNTR_ID is our unique value.

Next, instead of setting our counter variable at the HEAD REPORT section, we perform the initialization when the ENCNTR_ID changes.

```
HEAD ENCNTR_ID
      nCOUNT = 0
```

The rules for dynamically incrementing our structure with the ALTERLIST function remains the same; and assigning values is also the same. The only difference is that we are working in a deeper level of the structure.

The final step is to store our counter variable. This is done in the FOOT ENCNTR_ID section.

```
FOOT ENCNTR_ID
      rDATA->DATA[D.SEQ].ORDER_COUNT = nCOUNT
```

Outputting the Contents of a Record Structure

Once you have collected all the information you need into your record structure, the final step is using it in a report. Following our example above, we will continue by showing you how to output the results of our record structure.

Outputting a record structure is a simple task. It is performed primarily by writing a SQL statement against the DUMMYT table.

```
SELECT
      NURSE_UNIT                = rDATA->DATA[D.SEQ].NURSE_UNIT,
      ROOM                      = rDATA->DATA[D.SEQ].ROOM,
      BED                       = rDATA->DATA[D.SEQ].BED,
      PATIENT_NAME              = rDATA->DATA[D.SEQ].PATIENT_NAME,
      MRN                       = rDATA->DATA[D.SEQ].MRN,
      FIN_NBR                   = rDATA->DATA[D.SEQ].FIN_NBR,
      ORDER_COUNT               = rDATA->DATA[D.SEQ].ORDER_COUNT
FROM (DUMMYT                    D WITH SEQ=VALUE(rDATA->COUNT))
ORDER NURSE_UNIT, PATIENT_NAME
HEAD NURSE_UNIT
      COL  0, “NURSE UNIT: “, NURSE_UNIT , ROW + 2
DETAIL
      ; Print Patient Demographic Data

      COL 0, PATIENT_NAME
      COL 40, ROOM
      COL 50, BED
      COL 60, MRN
      COL 72, FIN_NBR
      ROW + 1
      ; Print Orders
      FOR (nLOOP = 1 TO ORDER_COUNT)
            IF (nLOOP = 1)
                  COL 5, “Orders: “
            ENDIF
            COL 15, rDATA->DATA[D.SEQ].ORDERS[nLOOP].ORDER_DT_TM
            COL 35, rDATA->DATA[D.SEQ].ORDERS[nLOOP].ORDERABLE
            ROW + 1
      ENDFOR
WITH NOCOUNTER
```

While not necessary, the first thing I like to do is to alias the fields in my record structure. I find it much easier to read NAME_FULL_FORMATTED than rDATA->DATA[D.SEQ].NAME_FULL_FORMATTED in the report writer section of the code.

So, immediately after my SELECT statement, I have aliased all of the fields I am capable of aliasing.

e.g.

```
NURSE_UNIT              = rDATA->DATA[D.SEQ].NURSE_UNIT,
      ROOM                    = rDATA->DATA[D.SEQ].ROOM,
      BED                     = rDATA->DATA[D.SEQ].BED,
etc..
```

You may notice that I didn’t alias the orders. This is because each record of the rDATA->DATA structure will have a different number of orders associated with them. Patient A may have 5 orders, Patient B may have 20. Aliasing the orders at this level would be impossible. However, I will show you shortly how to get around this limitation with a simple loop.

After aliasing our fields I choose the table I wish to perform my query against. Since all of our data is in our record structure, we only need to use the DUMMYT table. This is the same syntax we used when collecting our MRN/FIN # earlier in the example.

```
FROM (DUMMYT                  D WITH SEQ=VALUE(rDATA->COUNT))
```

Since I have aliased my fields, I can easily sort the report output on those fields. For my example, I am sorting the report by Nurse Unit followed by Patient Name.

```
ORDER NURSE_UNIT, PATIENT_NAME
```

With the exception of the orders, printing my report fields is the same as any other database field. I just use the alias names I have created.

e.g.

```
COL 0, PATIENT_NAME
```

As you remember, our orders have been defined three levels deep in our report. Printing these values in the report writer section of our report requires an additional step. Instead of just aliasing the fields like we did with our base information (Name, MRN, etc.), we need to use a FOR loop to iterate through all of the elements in the ORDERS branch of the structure.

The following code (which was also shown above) does just this.

```
; Print Orders
FOR (nLOOP = 1 TO ORDER_COUNT)
      IF (nLOOP = 1)
            COL 5, “Orders: “
      ENDIF
      COL 15, rDATA->DATA[D.SEQ].ORDERS[nLOOP].ORDER_DT_TM
      COL 35, rDATA->DATA[D.SEQ].ORDERS[nLOOP].ORDERABLE
      ROW + 1
ENDFOR
```

The only things to note here is that nLOOP is assigned a value from 1 to ORDER_COUNT (ORDER_COUNT is actually an alias of rDATA->DATA[D.SEQ].ORDER_COUNT).

When printing the field, you need to specify the entire field.

e.g.

```
rDATA->DATA[D.SEQ].ORDERS[nLOOP].ORDER_DT_TM
```

Summary

Mastering record structures is quite simple and over a short period of time, you will quickly realize how much easier your code becomes to read, debug and maintain. Additionally you benefit from a huge performance boost in the run time of your reports.

Understanding the Cerner Data Model

Cerner is a Transactional System

The Cerner Data Model was developed to quickly retrieve data into the front end applications. This sometimes makes report development a little difficult and we a required to think in terms of the front end when searching for our report data.

The majority of the reports we write detail data in a horizontal row. We may have patient identifying data such as name or MRN followed by other clinical data such as lab results. In terms of our report, everything is contained on one printed row.

However, the Cerner Data Model is designed so small efficient transactions can be passed between the front end applications and the database. For example, when examining lab results in PowerChart, the first step is to find your patient. At that time the basic patient information is loaded and placed in the Demog Bar at the top of the screen. When you click on the labs tab, the appropriate lab information for the default date range is loaded.

When developing your reports, you can use record structures and simple queries to mimic how the front end would retrieve the data. Once you have collected all of your report data into a record structure, you then present it as a horizontal row on your report.

By using this method, you no longer need to try and join unrelated data in a single select. Instead, if your report calls for collecting data from multiple sources, you do it separately the same way the front end does. This results in faster execution times and easier to maintain code.

For example, if we had a report that showed current patients along with their name, MRN, FIN #, Latest Weight and Height along with their next scheduled appointment, we could use a series of DONTCARE joins and come up with the data we need. We would likely have more than one row return for the patient and would have to code to accommodate it.

However, if we break the SELECT statement into multiple task oriented transactions, we could have separate SELECT statements that collect our patient population, alias records (MRN, FIN), Clinical Events (Height and Weight) and appointment information. Once collected, we would print our information out on our report.

By using this technique, if you need to modify a section (for example the clinical events), none of the other data collection pieces of code would need any modifications and would continue to run as is.

Cerner Table Naming Conventions

Cerner has somewhat been consistent in naming their tables in their Data Model. Table names do give some information on what contents may lie within. For example, the PERSON table contains information about people. The ENCOUNTER table contains encounters (or patient visit information).

Cerner has a number of primary tables which have additional related tables. Typically, the related tables use the name of the primary table in the table name along with an identifier. In some cases, a short formed version of the table name may be used.

Examples:

Primary Table: PERSON
Related Tables: PERSON_INFO, PERSON_PATIENT, PERSON_ALIAS

Primary Table: CLINICAL_EVENT
Related Tables: CE_BLOB, CE_EVENT_NOTE, CE_CODED_RESULT

Id Fields in Related Tables

When joining tables in your SELECT statements, it is always ideal to try and connect tables with a primary key where possible. Tables related to the primary tables above typically have a field that contains the primary key field from the primary table.

For example, the PERSON table has a primary key field called PERSON_ID. All of the PERSON_tablename tables have a field called PERSON_ID which ties the table to the PERSON table. Other tables (which are not directly related to the primary table) may also have a field that contains this key which will let you join related data together.

For example, the ENCOUNTER table contains patient visits. Rather than repeat all the PERSON information on the ENCOUNTER table, a field called PERSON_ID exists which can be used to join to the PERSON table or any of the other related tables (e.g. PERSON_PATIENT).

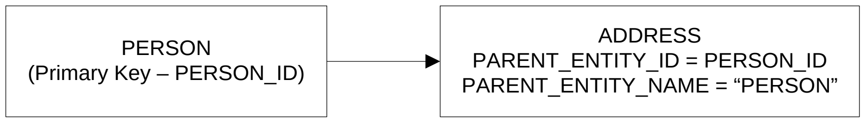
PARENT_ENTITY_ID

Some tables such as the ADDRESS and PHONE tables contain details for more than one parent table. For example, the ADDRESS table contains addresses for PERSON, LOCATION, ORGANIZATION and other types of data.

Rather than creating PERSON_ADDRESS, LOCATION_ADDRESS, ORGANIZATION_ADDRESS tables, Cerner created a single ADDRESS table to service all address needs. To accommodate the primary key from each of the primary tables, Cerner also created a field called PARENT_ENTITY_ID. This field contains key values from a number of sources (e.g. PERSON_ID, ORGANIZATION_ID, etc).

To help identify which table the key belongs to, the ADDRESS table also contains a field called PARENT_ENTITY_NAME. This table simply lists the name of the table for the key (e.g. PERSON, ORGANIZATION, ENCNTR_PLAN_RELTN, etc).

For example:


PERSON
(Primary Key – PERSON_ID)
ADDRESS
PARENT_ENTITY_ID = PERSON_ID
PARENT_ENTITY_NAME = "PERSON"

There are 240 tables in Cerner that use PARENT_ENTITY_ID fields. Thankfully there are only a handful that are typically used in CCL reports.

DBA_TAB_COLUMNS and DBA_TABLES

There are two tables in Cerner that make discovering the contents of tables much easier. These are the DBA_TAB_COLUMNS and DBA_TABLES tables.

The DBA_TABLES table is helpful when trying to determine what tables exist for reporting purposes. For example, if you wanted to see a list of all the tables that start with the word PERSON, you could use the following SELECT statement:

```
SELECT * FROM DBA_TABLES WHERE TABLE_NAME=”PERSON*”
```

The DBA_TAB_COLUMNS table is even more powerful. It can be used to show all the tables in Cerner that contain a specific fieldname. Some examples are:

```
SELECT * FROM DBA_TAB_COLUMNS WHERE COLUMN_NAME=”EVENT_ID”
SELECT * FROM DBA_TAB_COLUMNS WHERE COLUMN_NAME=”*NAME*”
SELECT * FROM DBA_TAB_COLUMNS WHERE COLUMN_NAME IN (“EVENT_ID”,”TASK_ASSAY_CD”)
```

Performance Testing and Tuning CCL Scripts

Determining the Efficiency of a CCL Script with DM_SQLPLAN

Every CCL script should be tested for performance issues. Doing so requires a minimal knowledge of Oracle as well as an understanding of how indexes work. Initially testing the performance of a CCL script requires the use of the back-end tool DM_SQLPLAN.

The DM_SQLPLAN tool reports how Oracle processes a CCL script. To use DM_SQLPLAN, access CCL from the Cerner back-end and complete the following steps.

1. From the CCL prompt, type in **DM_SQLPLAN GO**

2. The first prompt asks you to enter the number of queries to be analyzed. The default is 25. In most cases this value is sufficient. You would only change the default if you have more than 25 SELECT statements in your CCL script. If you are happy with the default, press the enter key.

3. When prompted to enter the script name, type in the object name of your script. The object name is the name you gave your program in the CREATE PROGRAM *yourprogramname*:DBA line of your CCL script. When you have finished entering the name, press the enter key.

4. The final prompt asks you to enter the sort criteria. Typically you would leave the default as 1 (Buffer Gets), but you can change the value to whatever suits your needs. I recommend using 1 (Buffer Gets) or 3 (Buffer Ratio) as they will cause the bad performing SELECT statements to show at the top of the report.

The following example shows what an inefficient CCL script looks like in DM_SQLPLAN.

Section 1 – The Plan Statement

The Plan statement simply shows how the CCL optimizer converted your CCL code into something Oracle can read. In most cases, you can skip right over this section unless you have an interest in how it is converted.

```
Plan statement for: 16972386       Hash Value:  4219421594.00

 SELECT  /*+  CCL<LP_BRONC_BRONC_W_BIOP:S9999:O1> */ CE.TASK_ASSAY_CD,CE.VERIFIED_DT_TM,P.NAME_FULL_FORMATTED,P.BIRTH_DT_TM,P
.SEX_CD,E.LOC_NURSE_UNIT_CD,E.LOC_ROOM_CD,E.LOC_BED_CD,E.REASON_FOR_VISIT,EA.ALIAS,EA2.ALIAS,PR.NAME_FULL_FORMATTED,PR2.NAME_
FULL_FORMATTED FROM  PRSNL PR2 ,PRSNL PR ,ENCNTR_PRSNL_RELTN EPR ,ENCNTR_ALIAS EA2 ,ENCNTR_ALIAS EA ,ENCOUNTER E ,PERSON P ,C
LINICAL_EVENT CE  WHERE CE.VERIFIED_DT_TM BETWEEN  :1    AND  :2    AND (CE.TASK_ASSAY_CD =     :3    OR CE.TASK_ASSAY_CD =
   :4   ) AND P.PERSON_ID =    CE.PERSON_ID AND E.PERSON_ID =    P.PERSON_ID AND EA.ENCNTR_ID =    E.ENCNTR_ID AND EA.ENCNTR_
ALIAS_TYPE_CD =     :5    AND EA.ACTIVE_IND =     :6    AND EA.END_EFFECTIVE_DT_TM >  :7    AND EA2.ENCNTR_ID =    EA.ENCNTR_
ID AND EA2.ENCNTR_ALIAS_TYPE_CD =     :8    AND EPR.ENCNTR_ID =    EA2.ENCNTR_ID AND  (EPR.ENCNTR_PRSNL_R_CD =     :9    OR E
PR.ENCNTR_PRSNL_R_CD =     :10  )  AND PR.PERSON_ID =    CE.PERFORMED_PRSNL_ID AND PR2.PERSON_ID =    CE.VERIFIED_PRSNL_ID
                                                                 Statistics
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
--------
```

Section 2 – The Statistics

The statistics section is helpful in determining how many resources your SELECT statement is using. There are three fields you should be concerned with in the statistics. The Buffer Gets represents the total number of buffer gets recorded against the execution of the report. Executions represent the number of times the SELECT statement has been run and Buffer Ratio represents the resources used for a single execution of the SELECT statement.

When analyzing a SELECT statement, DBA's typically focus on the Buffer Gets. They see the total number of resources being used for a SELECT statement as being the most critical factor. Through years of experience dealing with CCL reporting, I argue that the Buffer Ratio is really the most important number to be concerned with. If you have a total of 10,000 Buffer Gets and 100 executions, your ratio is really only 100 Buffer Gets and the report is quite efficient. However, there is likely a need to examine why the report has been run 100 times.

In the example below, we have one execution, so the ratio and the buffer gets are the same. In this example, 43,127 Buffer Gets is quite high. The only reason I know this is that the report only returned a single page of data. If I had run a report that resulted in 10,000 records being printed, 43,127 Buffer Gets would be quite acceptable.

```
Buffer Gets:                  Executions:                  Buffer Ratio:              Disk Reads:            Disk Ratio:
First Load Time:                   Score:               Hash Value:       Sharable Mem:    Optimizer Mode:            CPU
Time(sec):
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
--------
      43127.00                       1.00                    43127.00                      40043.00               40043.00
2007-04-26/12:45:38               8051927.00             4219421594.00         93209.00    RULE
1.96
```

Section 3 – The SQL Plan Info

The third section is perhaps the most important. The SQL Plan Info section shows you the order in which your tables are placed as well as which index is used. In this example, we have a major problem with our SELECT statement because the CLINICAL_EVENT table is doing a FULL table scan.

Our first goal is to eliminate the FULL table scan and use an indexed field to help correct the performance of this report.

On the report below, the following pieces of information are important to know:

Options describes the type of options in relation to the Operation taking place. The values you are concerned with are: UNIQUE SCAN, RANGE SCAN, FULL. Oracle will attempt to choose the appropriate index based on the order of these options. If your SELECT statement filters on a number of fields, an index will automatically be chosen in the order of UNIQUE SCAN, RANGE SCAN and finally FULL scan.

An example of a UNIQUE SCAN is if you are filtering on the ENCNTR_ID equalling the ENCNTR_ID on another joining table. A RANGE scan would be a range of code values or possibly a date range.

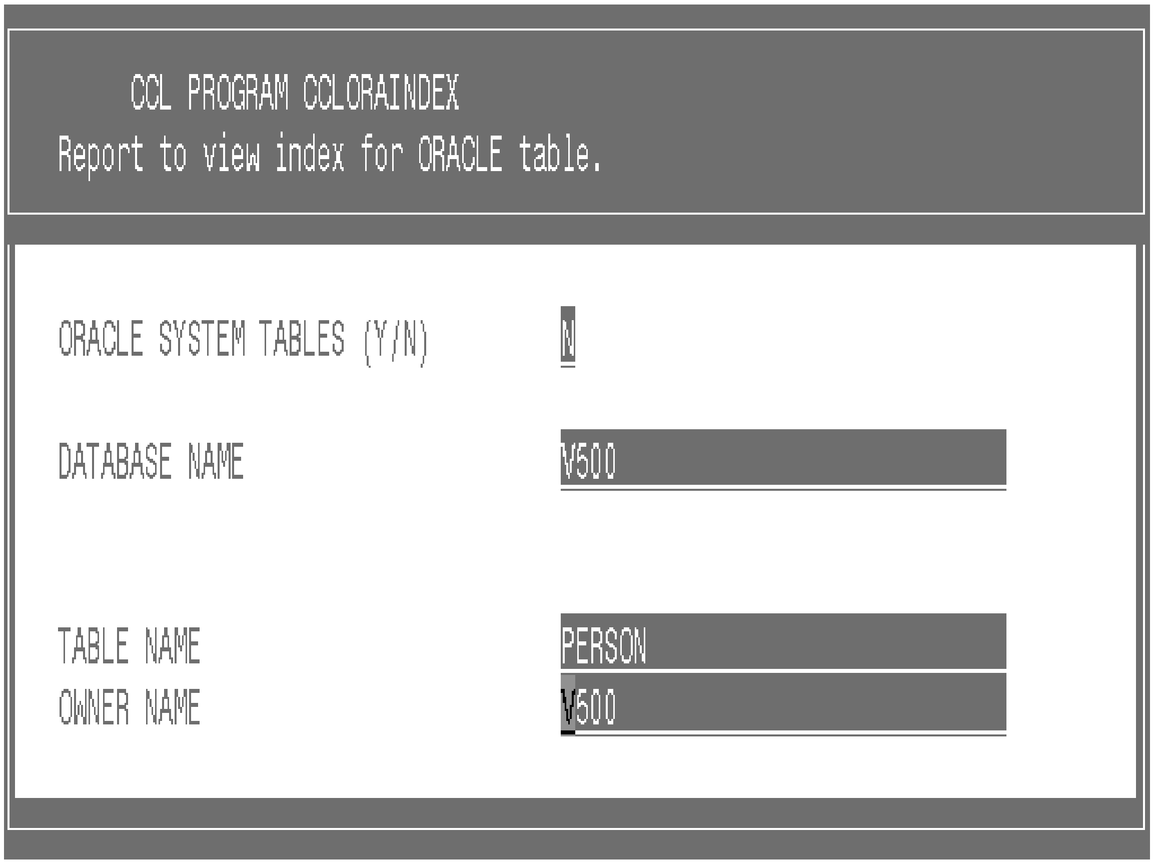
Object Name represents the name of the table in question or the name of the index used in the SELECT statement. In the example below, PERSON is the name of a table where XPKPERSON is the index that was used when retrieving the data from the PERSON table.

Index Columns shows the fields in the index were used in the SELECT statement. Examining this column against your SELECT statement is very helpful in determining if your query is using the indexed fields you had expected.

```
                                                                SQL Plan Info
-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
--------
Operation                      Options                        Object Name                    Index Columns
-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
--------
SELECT STATEMENT
TABLE ACCESS                   BY INDEX ROWID                 ENCNTR_PRSNL_RELTN
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
TABLE ACCESS                   FULL                           CLINICAL_EVENT
TABLE ACCESS                   BY INDEX ROWID                 PERSON
INDEX                          UNIQUE SCAN                    XPKPERSON                      PERSON_ID
TABLE ACCESS                   BY INDEX ROWID                 PRSNL
INDEX                          UNIQUE SCAN                    XPKPRSNL                       PERSON_ID
TABLE ACCESS                   BY INDEX ROWID                 PRSNL
INDEX                          UNIQUE SCAN                    XPKPRSNL                       PERSON_ID
TABLE ACCESS                   BY INDEX ROWID                 ENCOUNTER
INDEX                          RANGE SCAN                     XIE1ENCOUNTER                  PERSON_ID,BEG_EFFECTIVE_DT_TM
TABLE ACCESS                   BY INDEX ROWID                 ENCNTR_ALIAS
INDEX                          RANGE SCAN                     XIE2ENCNTR_ALIAS               ENCNTR_ID
TABLE ACCESS                   BY INDEX ROWID                 ENCNTR_ALIAS
INDEX                          RANGE SCAN                     XIE2ENCNTR_ALIAS               ENCNTR_ID
INDEX                          RANGE SCAN                     XIE5ENCNTR_PRSNL_RELTN
ENCNTR_ID,PRSNL_PERSON_ID,EXPIRATION_IND
```

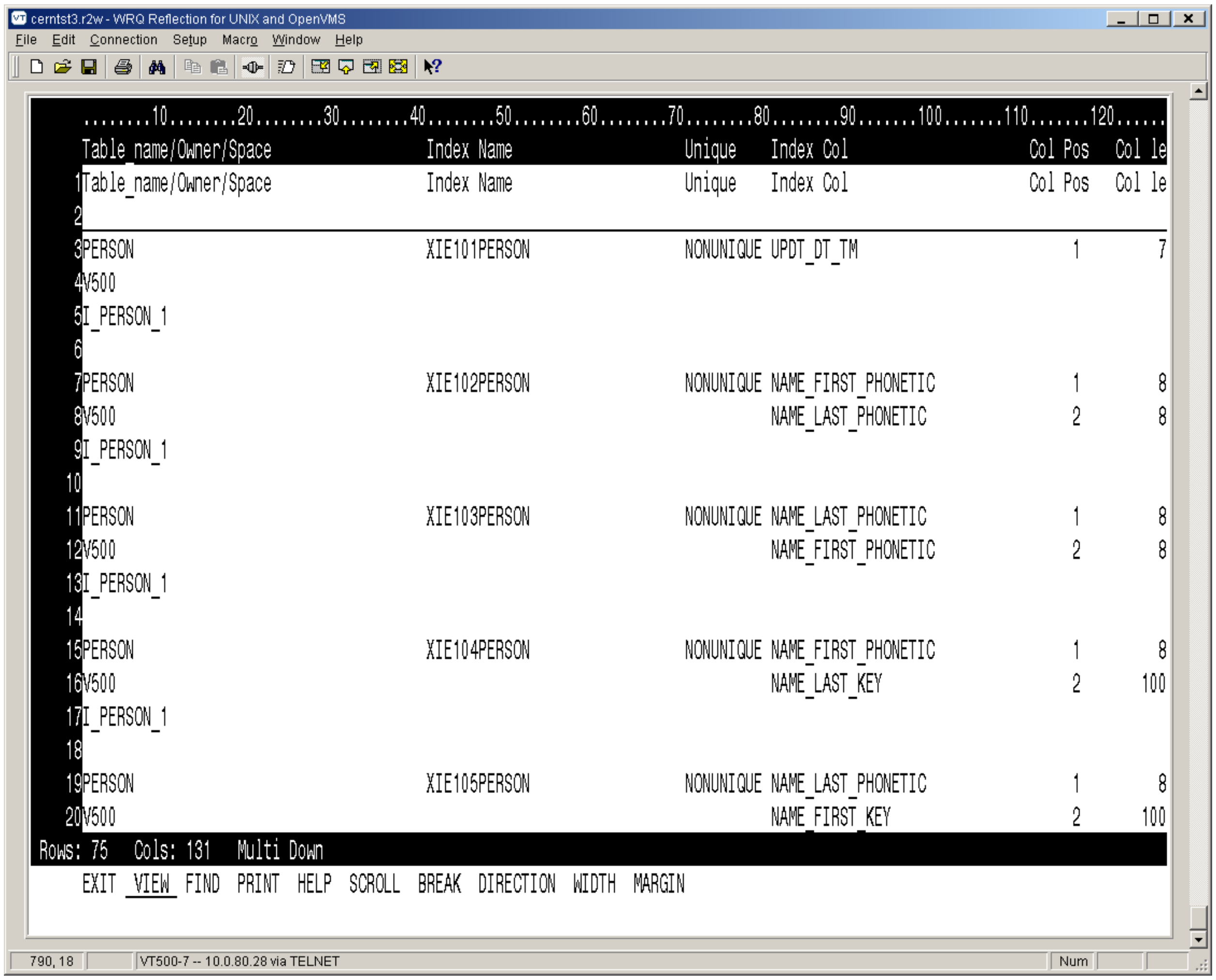
CCLORAINDEX

CCLORAINDEX is a handy little back-end tool that will show you a list of all the indexes in a table. Simply type in CCLORAINDEX GO from the CCL command line and fill in the prompts. All of the prompts with the exception of TABLE NAME should remain with their default values. For table name, type in the name of the table you wish to see the indexes in.



```
     CCL PROGRAM CCLORAINDEX
 Report to view index for ORACLE table.

 ORACLE SYSTEM TABLES (Y/N)        N

 DATABASE NAME                     V500


 TABLE NAME                        PERSON
 OWNER NAME                        V500
```

Once the prompts have been completed, you will be presented with a report of the available indexes on the table. Unique and non-unique indexes are identified on the report.


cernst3.r2w - WRQ Reflection for UNIX and OpenVMS
File Edit Connection Setup Macro Window Help

```
     ........10........20........30........40........50........60........70........80........90.......100.......110.......120......
     Table_name/Owner/Space                  Index Name                Unique   Index Col                      Col Pos   Col le
    1Table_name/Owner/Space                  Index Name                Unique   Index Col                      Col Pos   Col le
    2
    3PERSON                                  XIE101PERSON              NONUNIQUE UPDT_DT_TM                          1         7
    4V500
    5I_PERSON_1
    6
    7PERSON                                  XIE102PERSON              NONUNIQUE NAME_FIRST_PHONETIC                 1         8
    8V500                                                                        NAME_LAST_PHONETIC                  2         8
    9I_PERSON_1
   10
   11PERSON                                  XIE103PERSON              NONUNIQUE NAME_LAST_PHONETIC                  1         8
   12V500                                                                        NAME_FIRST_PHONETIC                 2         8
   13I_PERSON_1
   14
   15PERSON                                  XIE104PERSON              NONUNIQUE NAME_FIRST_PHONETIC                 1         8
   16V500                                                                        NAME_LAST_KEY                       2       100
   17I_PERSON_1
   18
   19PERSON                                  XIE105PERSON              NONUNIQUE NAME_LAST_PHONETIC                  1         8
   20V500                                                                        NAME_FIRST_KEY                      2       100
Rows: 75   Cols: 131   Multi Down
     EXIT  VIEW  FIND  PRINT  HELP  SCROLL  BREAK  DIRECTION  WIDTH  MARGIN
```

Forcing an Index

Sometimes no matter how hard you try to organize your SELECT statement, the index you want to use is ignored and a less than ideal index is chosen instead.

There are two ways to force an index to be used. The first is to convert the field being selected in your index to a number. This is done by adding zero to your where clause.

For example:

```
WHERE SA.APPT_TYPE_CD+0 = 33944
```

The second method is to use the ORAHINT control option. To use ORAHINT, you need to know the name of the index your wish to force. The index name can be found using CCLORAINDEX. The ORAHINT is added to the WITH statement at the end of your query.

```
SELECT .....
FROM TABLE_NAME T
PLAN T
WHERE T ....
WITH ORAHINT("INDEX (T XIE_INDEX_NAME)")
```

Using Include Scripts to Promote Code Re-Use

Often we find that much of the code we write is repeated from program to program. For example, your site will have many reports that are based on the current census. Those same reports may also use the same types of data from other Cerner modules. It is not uncommon to have a report that is based on current census that shows all patients with labs ordered and another report that shows the same census with labs + radiology orders.

You can code each of these reports as separate reports (and in many cases it is recommended), but if you find you have a large number of reports that are very similar in data collection, having code that you can re-use will not only save time in development, but make future changes easier to maintain.

This is accomplished with an include script. An include script is nothing more than a text file that contains CCL code.

For example, the following two programs will do exactly the same thing. The only differences are that the second script uses an include file.

Source code for test1.prg

```
DROP PROGRAM TEST1:DBA GO
CREATE PROGRAM TEST1:DBA

SELECT * FROM ENCOUNTER WHERE REG_DT_TM > CNVTDATETIME(CURDATE-1,0)

END GO
```

Source code for test2.prg

```
DROP PROGRAM TEST2:DBA GO
CREATE PROGRAM TEST2:DBA

%INCLUDE TEST2.INC

END GO
```

Source code for test2.inc

```
SELECT * FROM ENCOUNTER WHERE REG_DT_TM > CNVTDATETIME(CURDATE-1,0)
```

When TEST2.PRG is compiled, the line %INCLUDE TEST2.INC is replaced by all of the content in the TEST2.INC file. The final result is exactly the same as TEST1.PRG.

You can use include files for a number of purposes. For example, you may have a standard set of code values that you wish to have available to all of your programs. Instead of coding the lookups in every script, you can simply have a include file similar to the example below.

Script Name: common_code_values.inc

```
; ENCNTR_TYPE_CD
SET cvINPATIENT = UAR_GET_CODE_BY("DISPLAYKEY",71,"INPATIENT")
SET cvOUTPATIENT = UAR_GET_CODE_BY("DISPLAYKEY",71,"OUTPATIENT")
SET cvEMERGENCY = UAR_GET_CODE_BY("DISPLAYKEY",71,"EMERGENCY")

; ENCNTR_ALIAS_TYPE_CD
SET cvMRN = UAR_GET_CODE_BY("DISPLAYKEY",319,"MRN")
SET cvFIN_NBR = UAR_GET_CODE_BY("DISPLAYKEY",319,"FIN NBR")
```

Once you include this code into your main program file, all of the variables defined will be available for use.

e.g.

```
DROP PROGRAM TEST:DBA GO
CREATE PROGRAM TEST:DBA

%INCLUDE COMMON_CODE_VALUES.INC

SELECT * FROM ENCOUNTER WHERE REG_DT_TM > CNVTDATETIME(CURDATE, CURTIME3) AND
ENCNTR_TYPE_CD = cvEMERGENCY

END GO
```

As your needs grow, you can continue to add to your include script without affecting the main program.

You can also use include scripts to define custom subroutines which can be used in your scripts. For example, you may commonly have MRN as a prompt in your Explorer Menu reports. From this, you may wish to have a subroutine that returns the most current active ENCNTR_ID for that MRN which you will use in your select statements. Code to do this would be as follows:

Script Name: lookup_enc_by_mrn.inc

```
%INCLUDE COMMON_CODE_VALUES.INC

SET nENCNTR_ID = 0
```

```
SUBROUTINE LOOKUP_ENC_BY_MRN(cMRN)

      SELECT INTO “NL:”
            ENCNTR_ID         = E.ENCNTR_ID,
            REG_DT_TM         = FORMAT(E.REG_DT_TM,”YYYYMMDDHHMM;;Q”)
      FROM  ENCNTR_ALIAS            EA
            ENCOUNTER         E
      PLAN EA
            WHERE EA.ALIAS = cMRN
            AND EA.ENCNTR_ALIAS_TYPE_CD = cvMRN
            AND EA.ACTIVE_IND = 1
            AND EA.END_EFFECTIVE_DT_TM > CNVTDATETIME(CURDATE, CURTIME3)
      JOIN E
            WHERE E.ENCNTR_ID = EA.ENCNTR_ID
            AND E.ACTIVE_IND = 1
      ORDER REG_DT_TM
      DETAIL
            nENCNTR_ID = ENCNTR_ID
      WITH NOCOUNTER

END
```

You may have noticed that I included the common code value include script in this include file. I did this because I needed to make use of the cvMRN code value. Because of the code reuse offered by include scripts; I do not need to re-write the same common code over and over again.

Now, if I wanted to use this script, I might have the following code in my program:

```
DROP PROGRAM TEST:DBA GO
CREATE PROGRAM TEST:DBA

PROMPT “MRN:”=”x”

%INCLUDE LOOKUP_ENC_BY_MRN.INC

CALL LOOKUP_ENC_BY_MRN(VALUE($1))

SELECT * FROM ENCOUNTER WHERE ENCNTR_ID = nENCNTR_ID

END GO
```

Final Thoughts on Include Scripts

It’s possible to achieve a great deal of code re-use with simple include scripts. Any code you can place in your main CCL script can be placed in an include script. While it may be tempting to break your reports into many include scripts, I don’t recommend it. Doing so will lead to issues in being able to follow and read your code.

Instead, I suggest planning your include files so they can be used to save time but still keep your report code meaningful. Create code libraries that can be re-used as well as common lookups. These things save time and make your code easier to follow.

Layout Builder

Introduction to Layout Builder

The Layout Builder is a recent addition to CCL that allows for rapid development of visually stunning reports. Traditionally, CCL programmers had two methods of writing reports. They either created text only reports or PostScript reports using DIO commands.

While text reports are very simple to create, they leave much to be desired when the issue of formatting comes up. Special formatting such as shading and a variety of fonts are just not possible.

To create visually appealing reports, many CCL programmers use the DIO functions in CCL to create PostScript reports. This method gives the programmer the power and flexibility to create reports that make use of various fonts as well as shading, lines and barcodes. The only downside is that writing DIO PostScript reports is very time consuming.

Layout Builder lets you create the equivalent of a DIO PostScript report without all of the work. Instead of hand coding the coordinates of every piece of text, you simply drag and drop your text into place.

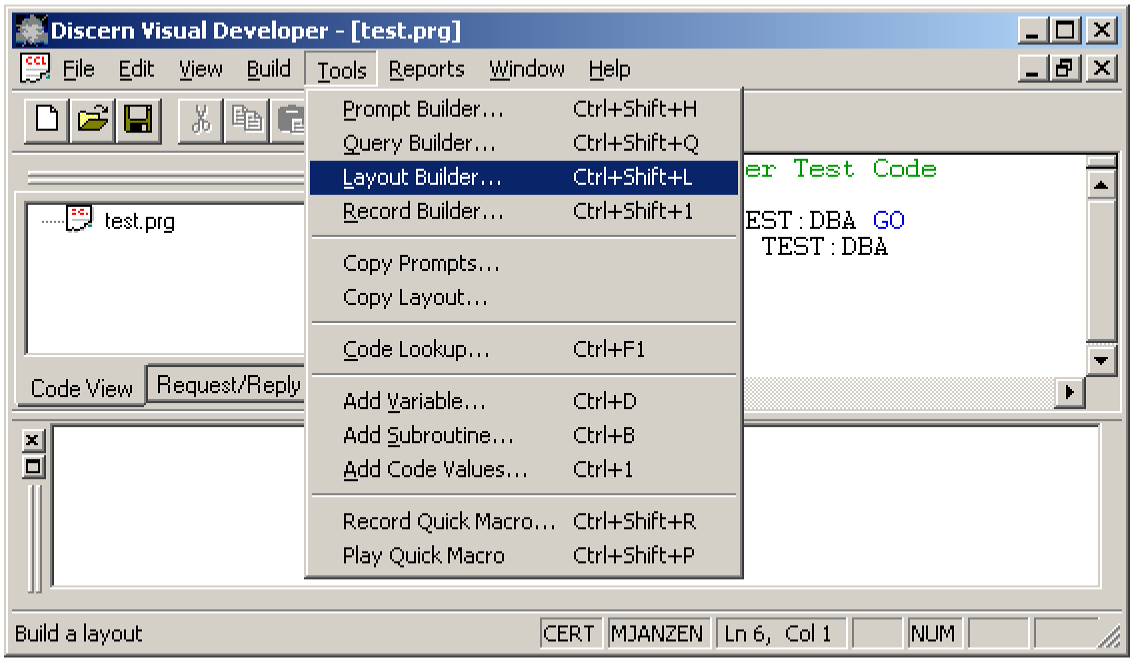
Using Layout Builder

Let’s step through working with Layout Builder by creating a simple example. Create a program in DiscernVisualDeveloper and call it test_yourinitials.prg. For my example I am going to simply call my script test.prg.

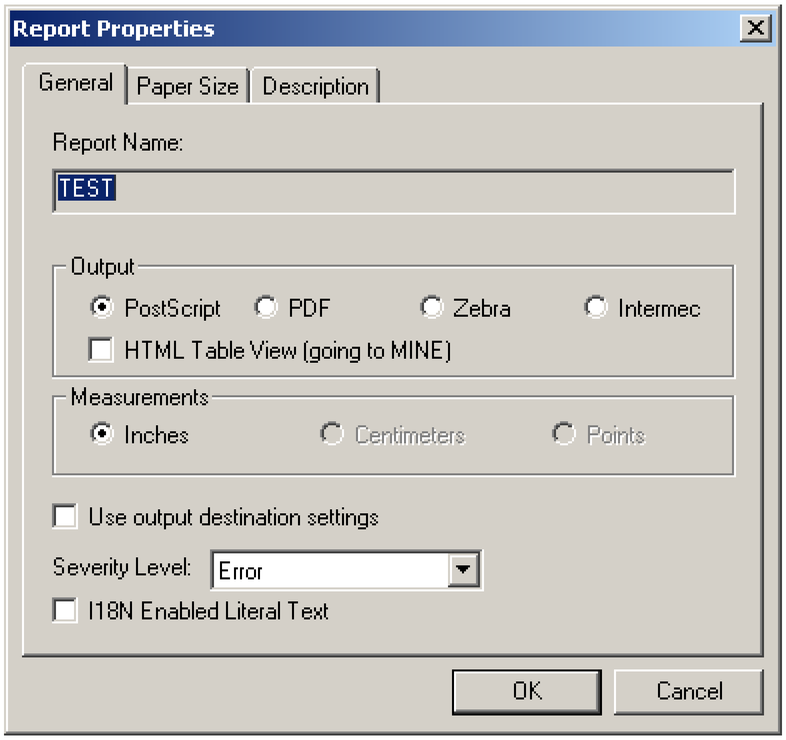
Start with the following basic code (don’t forget to add your initials in the DROP/CREATE lines).

```
DROP PROGRAM TEST:DBA GO
CREATE PROGRAM TEST:DBA

SELECT INTO “NL:”

      NURSE_UNIT          = UAR_GET_CODE_DISPLAY(E.LOC_NURSE_UNIT_CD),
      ROOM                = UAR_GET_CODE_DISPLAY(E.LOC_ROOM_CD),
      BED                 = UAR_GET_CODE_DISPLAY(E.LOC_BED_CD),
      PATIENT_NAME        = P.NAME_FULL_FORMATTED

FROM  ENCOUNTER           E,
      PERSON              P

PLAN  E
      WHERE E.REG_DT_TM > CNVTDATETIME(CURDATE-1, CURTIME3)
      AND   E.ACTIVE_IND = 1

JOIN  P
      WHERE P.PERSON_ID = E.PERSON_ID

END GO
```

Next, launch the Layout Builder by choosing Layout Builder from the Tools menu in Visual Developer.


Discern Visual Developer - [test.prg]
File Edit View Build Tools Reports Window Help
Prompt Builder... Ctrl+Shift+H
Query Builder... Ctrl+Shift+Q
Layout Builder... Ctrl+Shift+L
Record Builder... Ctrl+Shift+1
Copy Prompts...
Copy Layout...
Code Lookup... Ctrl+F1
Add Variable... Ctrl+D
Add Subroutine... Ctrl+B
Add Code Values... Ctrl+1
Record Quick Macro... Ctrl+Shift+R
Play Quick Macro Ctrl+Shift+P
test.prg
Code View
Request/Reply
er Test Code
EST:DBA GO
TEST:DBA
Build a layout
CERT
MJANZEN
Ln 6, Col 1
NUM

If it is your first time in Layout Builder, you will be greeted with the Report Properties screen. For a standard PostScript report, you can leave everything as shown and select the Ok button. However, if you wish to create a barcode label, you can do so by selecting either Zebra or Intermec from this screen (Choose based on the types of printer your organization uses).



Report Name: TEST

Output: [✓ PostScript] PDF Zebra Intermec

HTML Table View (going to MINE)

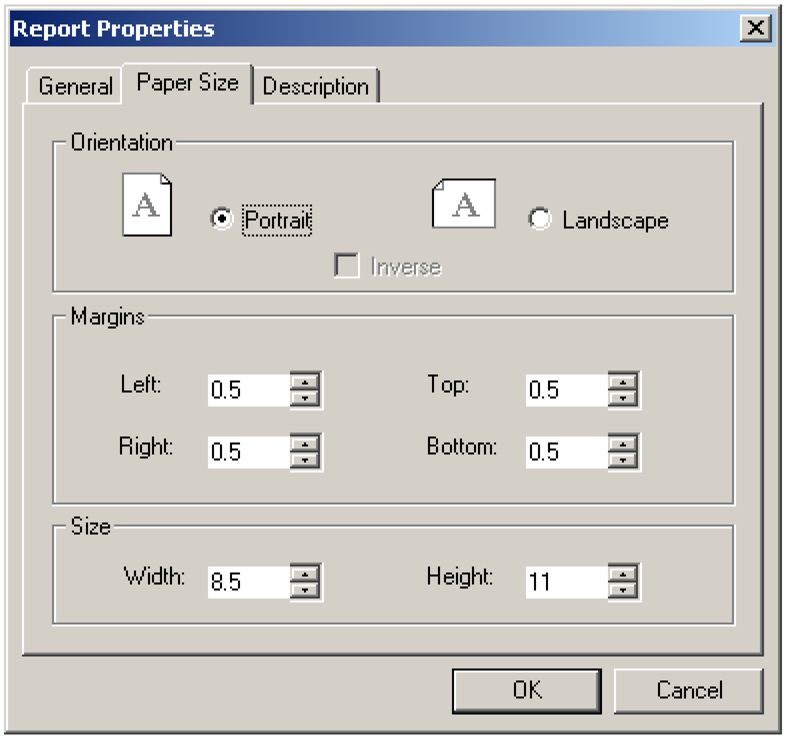
Measurements: [✓ Inches] Centimeters Points

Use output destination settings

Severity Level: Error

I18N Enabled Literal Text

The Paper size Tab on the Report Properties screen is something you will use on a frequent basis. This screen allows you the option of choosing whether a report is Portrait vs. Landscape and lets you set your page margins as well as page size.



Orientation: [✓ Portrait] Landscape
Inverse: ☐

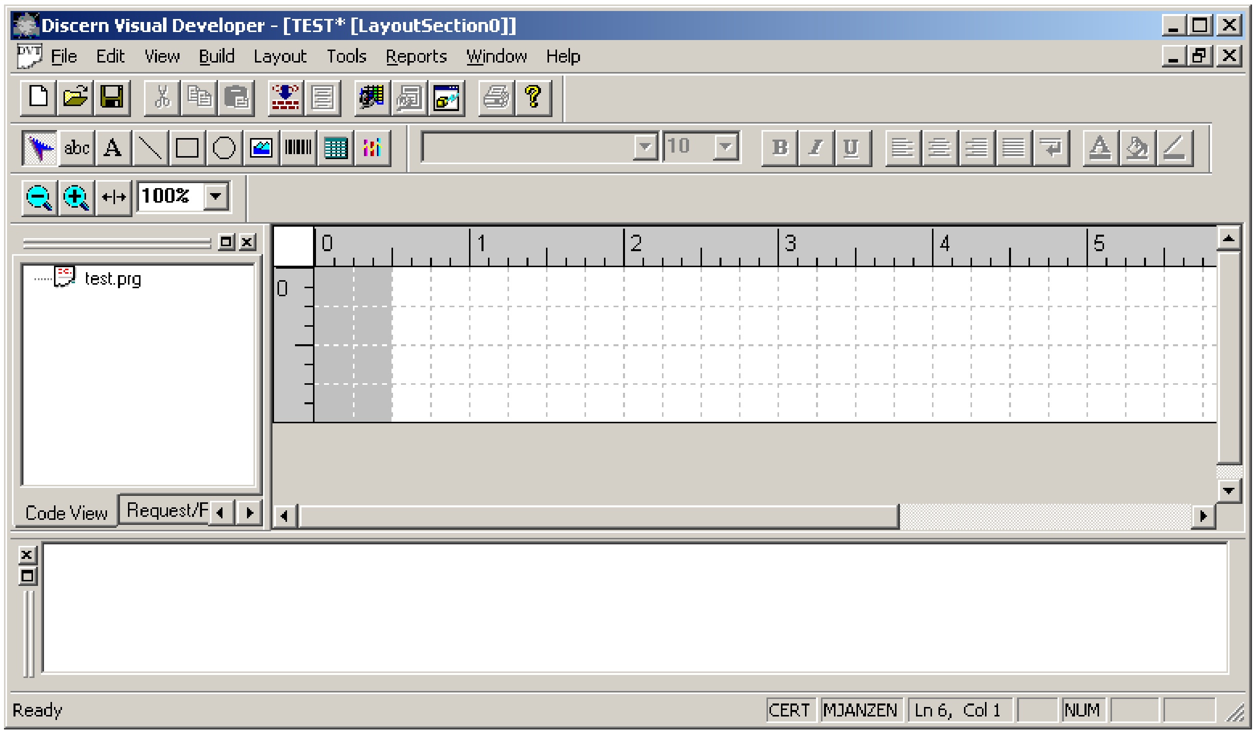
Margins
Left: 0.5
Top: 0.5
Right: 0.5
Bottom: 0.5

Size
Width: 8.5
Height: 11

For our test report, we will leave the Paper Size properties at their defaults.

If you need to return to the Report Properties screen at any time, it can be done by choosing The Properties followed by Report Properties from the Edit Menu in the Layout Builder.

Once you have set your Report Properties, you are greeted with the main Layout Builder workspace. The workspace will initially look like the following screen:


Discern Visual Developer - [TEST* [LayoutSection0]]
File Edit View Build Layout Tools Reports Window Help
abc
10
100%
test.prg
Code View
Request/F
Ready
CERT
MJANZEN
Ln 6, Col 1
NUM

The top bar of icons is the same standard set of icons seen when writing your code. You have your new, open, save, compile buttons.

The second menu bar is specific to the Layout Builder. Each of the icons is explained below.

The Select button is used to select objects that already exist in your workspace. By using this button, you will be able to click on objects to view and modify their properties.

abc The Label button is used to place text on your report that does not change. Examples of this would be field headers (e.g. Patient Name:)

A The Text button is used to place variables on your report. This is the object you would use to print calculations and field values from your SQL statements.

The Line button allows you the ability to place lines on your report. Lines can be vertical, horizontal or on left or right angles.

The Rectangle and Oval buttons are used to draw rectangle and circular shapes.

The Image button allows you the ability to add pictures to your reports. Common uses are to add your hospital logo to the report. The image must be stored on the same machine as your Cerner install to function.

The Barcode button gives you the ability to print a barcode on your report. CCL offers the ability to print barcodes on a PostScript report as well as the barcode printers (e.g. Intermec).

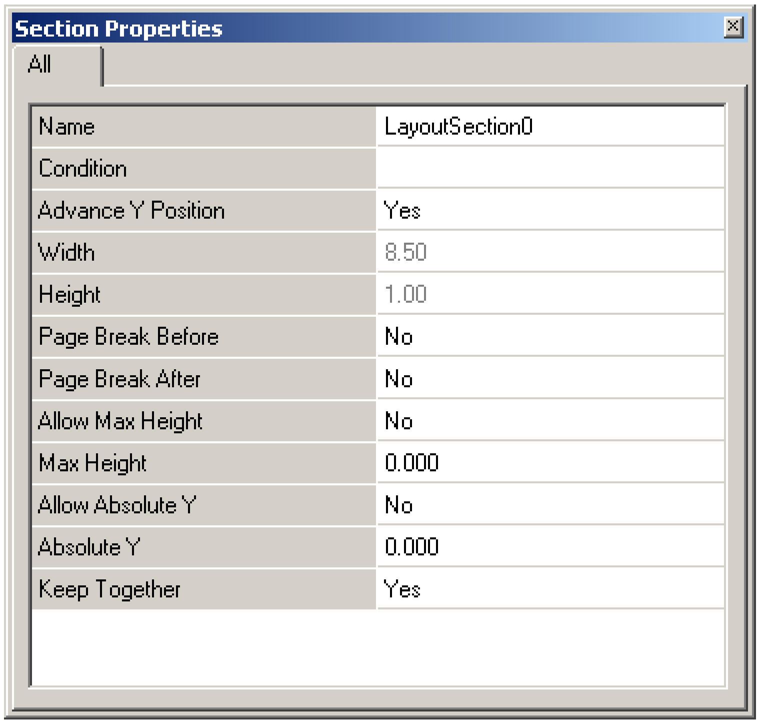
The Table and Graph buttons can be used to create tables and graphs on your report. Due to the complexity of developing reports that use tables and graphs, they are out of scope for this documentation and will not be covered.

Once you have familiarized yourself with the buttons, the next step is to get to know the workspace. On the main section of the screen, you will see a white area with grid lines covering it. This is the workspace where you will be adding items to your report.

This workspace is called a section. Your report can have many sections and most reports have at least two (a header and detail).

For this example, we are going to start by building our report header section. This is where we will put our report titles.

First we need to rename the section. Do this by choosing Properties followed by Section Properties from the Edit Menu. You will see the following window:

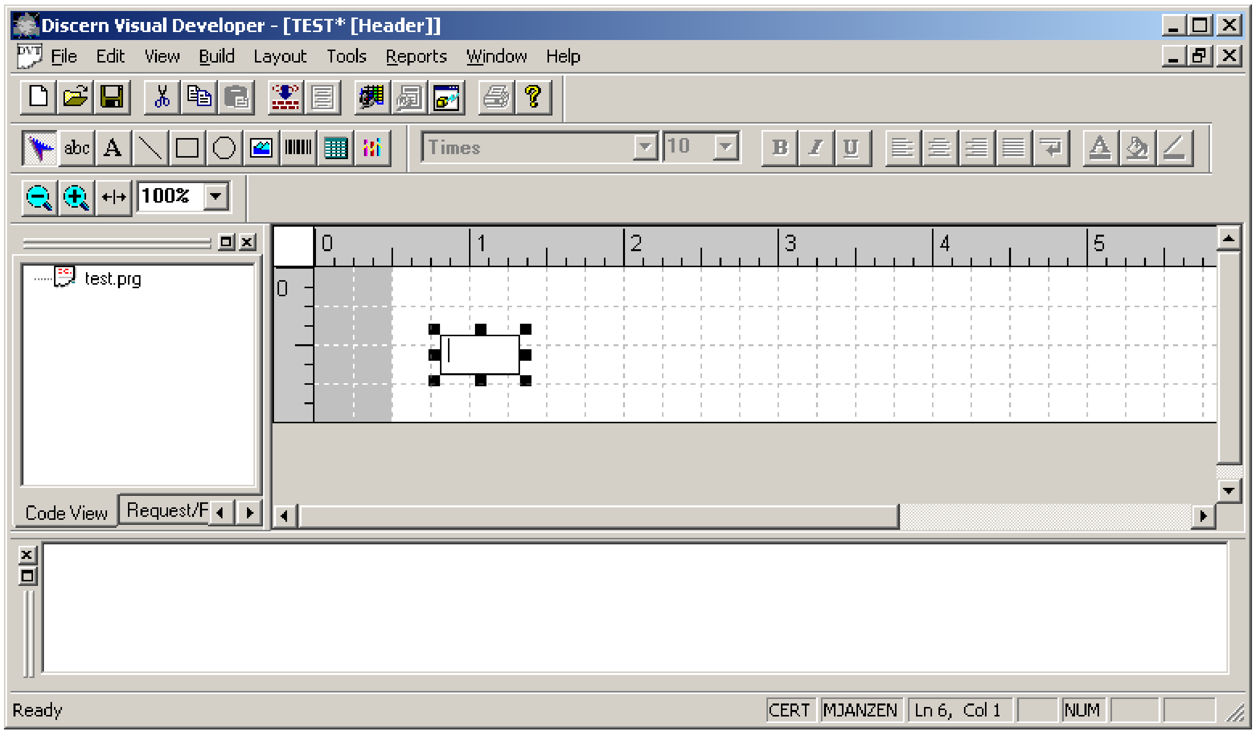


Name	LayoutSection0
Condition	
Advance Y Position	Yes
Width	8.50
Height	1.00
Page Break Before	No
Page Break After	No
Allow Max Height	No
Max Height	0.000
Allow Absolute Y	No
Absolute Y	0.000
Keep Together	Yes

Change the name to **Header** and close the window by click the X in the top right corner of the window. You will see that your changes have worked because the title bar of your screen will now show the name of your section.

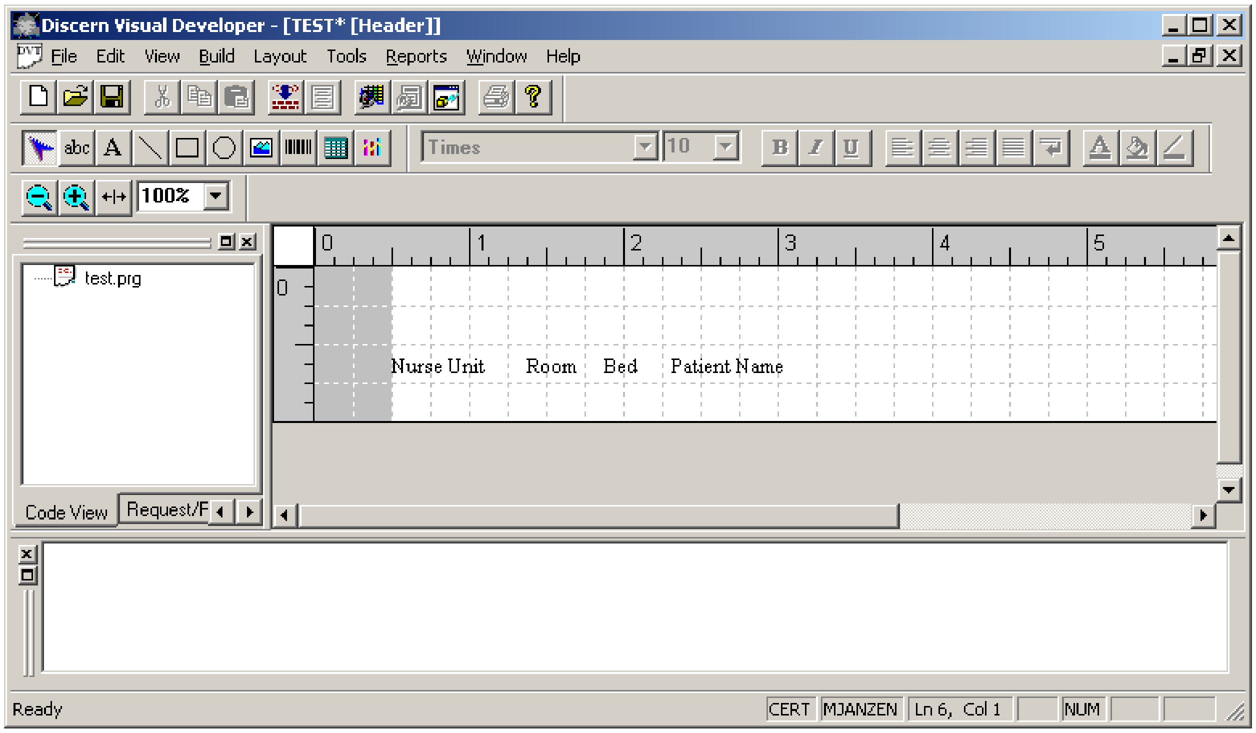
Discern Visual Developer - [TEST* [Header]]
File Edit View Build Layout Tools Reports Wind

Next, let's add some labels to our section. Do this by clicking the Label button abc followed by clicking anywhere in the section that you wish to add your label. You will see a box appear with a set of sizing handles.


Discern Visual Developer - [TEST* [Header]]
File Edit View Build Layout Tools Reports Window Help
Times
10
100%
test.prg
Code View
Request/F
Ready
CERT
MJANZEN
Ln 6, Col 1
NUM

At this point, simply type in the text you wish to place on your report. For this example, type in **Nurse Unit** and press Enter. The first thing you will notice is that the words Nurse Unit don't fit in the default box size. Use the sizing handles to re-size the box to your liking. This is done by left clicking on a sizing handle, holding down the left mouse button and dragging to the size you want.

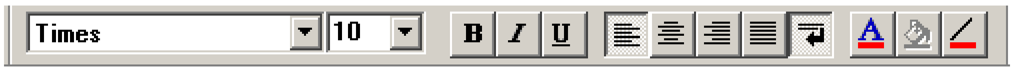
Once you are done, add three more labels called **Room**, **Bed**, and **Patient Name**. Feel free to move the labels around so they line up nicely.


Discern Visual Developer - [TEST* [Header]]
File Edit View Build Layout Tools Reports Window Help
Times
10
100%
test.prg
Nurse Unit
Room
Bed
Patient Name
Code View
Request/F
Ready
CERT
MJANZEN
Ln 6, Col 1
NUM

By now, your section should look similar to the example above. Next, let's pretty the section up a bit by bolding the labels and putting a line across the report.

You can do the next step either by single clicking and object or you can select multiple objects by holding down the left mouse button and dragging over the objects.

Select one or all of the fields now. You will notice that the screen has changed and enabled a new selection of buttons. These buttons allow you some formatting options such as choosing font, bolding, text justification and font colors.



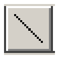
You can change the font by clicking the arrow to the right of the font name Times. Simply choose the font you desire. Next to the font name is the font size. Choose the size you wish your font to appear in.

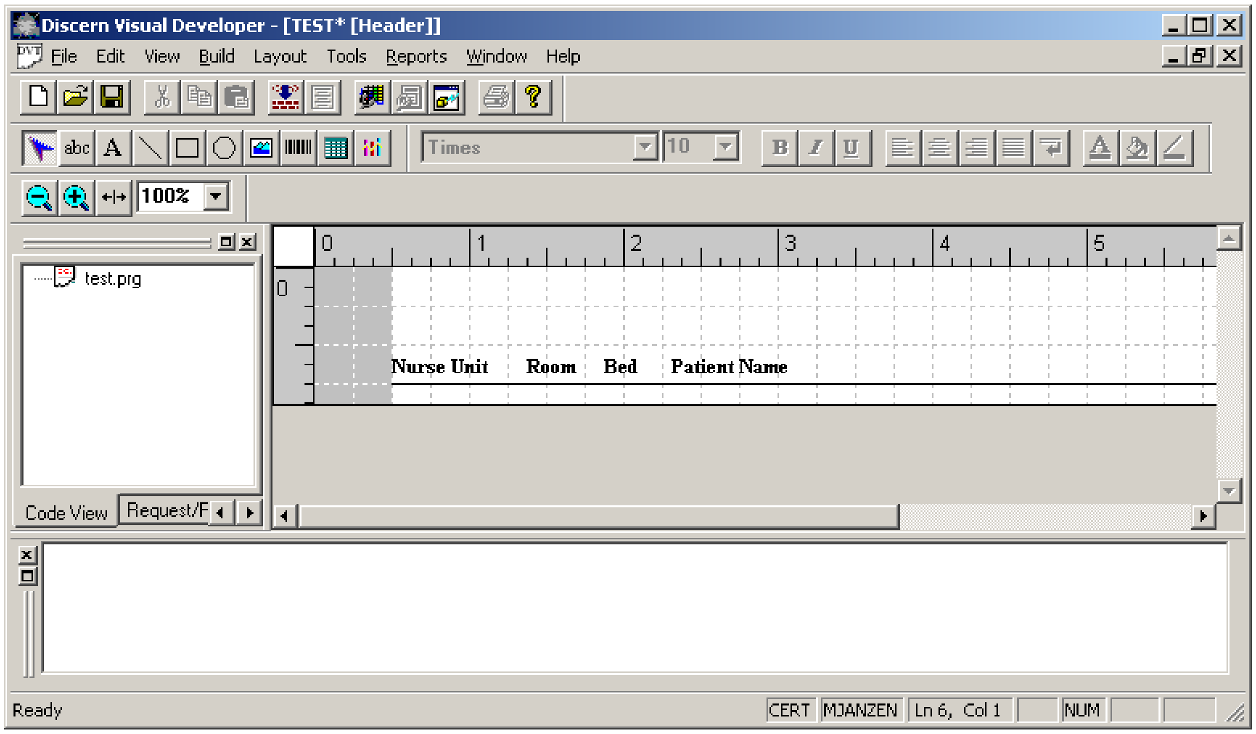
Next are three formatting buttons; Bold, Italic and Underline. These work the same as they do in all word processing applications.

The next group of buttons offers five text justification options. The first four are your typical left, center, right, justify options found in word processors. The fifth button (which is selected by default) indicates that text will wrap if it is too long to print in the width provided. If your box is tall enough, you can print output on multiple lines.

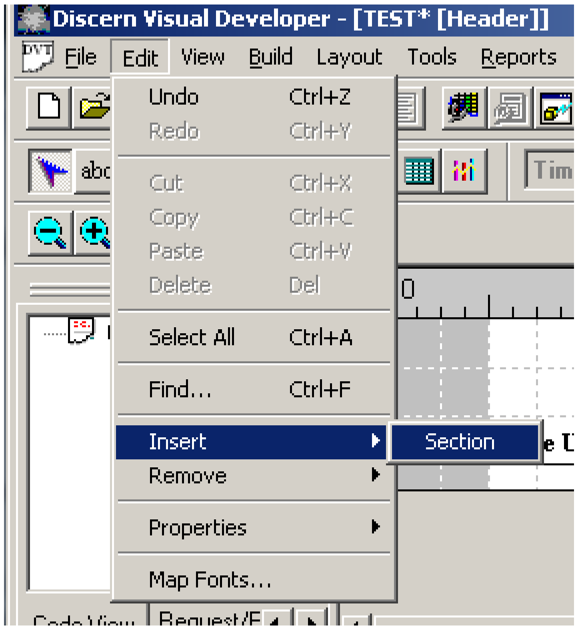
The final set of buttons is used to set Text Color, Back Color and Pen Color. Typically your reports will be using black for all of your content, however you may have a need to print color reports. Layout Builder gives you the ability to create reports in full color.

Continuing with our example, click the **Bold** button to bold your labels.

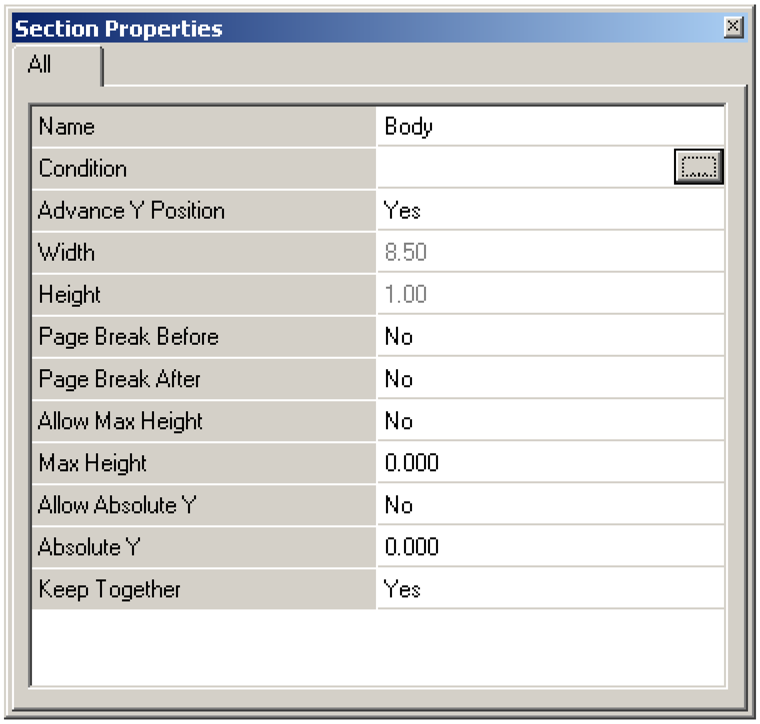
Next, click the Line button  and then move your mouse below the words Nurse Unit and left click the mouse. A short horizontal line will appear. Use the sizing handles to drag the line so it fills the entire width of the report. Your final result should look something like the section below.


Discern Visual Developer - [TEST* [Header]]
File Edit View Build Layout Tools Reports Window Help
Times
10
100%
test.prg
Nurse Unit
Room
Bed
Patient Name
Code View
Request/F
Ready
CERT
MJANZEN
Ln 6, Col 1
NUM

Next, we are going to create a section to show the details of our report. From the Edit menu, select Insert followed by Section.



You will be prompted with a new Section Properties window called LayoutSection1. Rename LayoutSection1 to Body. Be careful to not ever call the section Detail as Detail is a reserved Cerner keyword and will cause your program to fail.



Name	Body
Condition	
Advance Y Position	Yes
Width	8.50
Height	1.00
Page Break Before	No
Page Break After	No
Allow Max Height	No
Max Height	0.000
Allow Absolute Y	No
Absolute Y	0.000
Keep Together	Yes

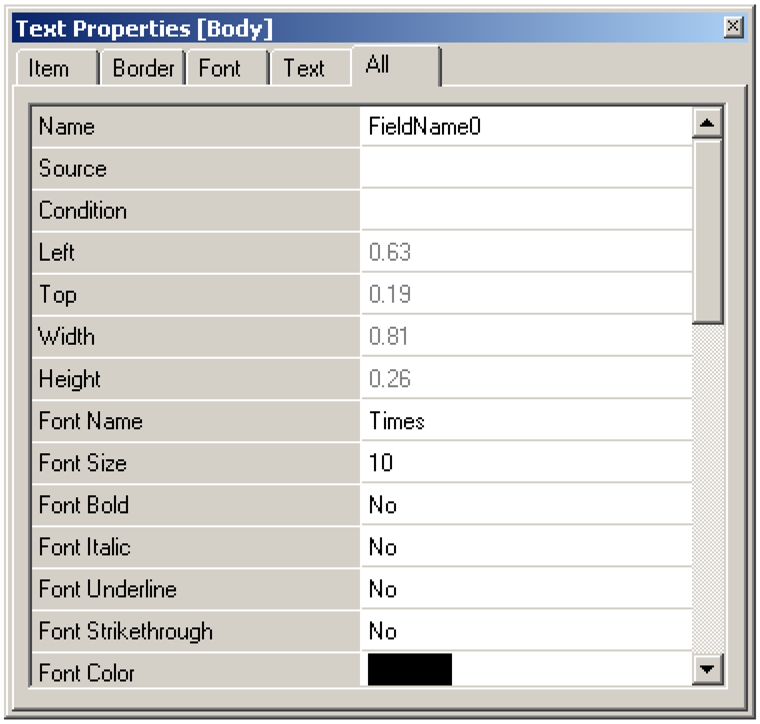
Once you have changed the name to Body, click the X in the right hand corner of the window and you will see that a new section has appeared below the first section.

Our next step is to start adding the fields from our SQL query to the report. If you look back at the code, you will see we aliased the following fields: NURSE_UNIT, ROOM, BED, PATIENT_NAME.

Start by clicking the Text icon A from the menu bar. Next, left click in the new section somewhere below the words Nurse Unit. Don't worry if you are not exactly where you want to be, you can reposition later by dragging your Text object.

You will see the words FieldNam appear in a box with sizing handles. Go ahead a resize this field to see that the name is actually FieldName0. Every time you add a Text object, this number will increment.

At this point, this field doesn't do anything useful. Double click on it so you can see the properties.



Name	FieldName0
Source	
Condition	
Left	0.63
Top	0.19
Width	0.81
Height	0.26
Font Name	Times
Font Size	10
Font Bold	No
Font Italic	No
Font Underline	No
Font Strikethrough	No
Font Color	■

In the Text Properties window, the three most important fields are the first three.

- **Name** represents the name of the field. This can be anything you want so long as it only contains text and numbers (e.g. NurseUnit). The name of the field must also be unique.
- **Source** is where you will determine what is printed in the field. Typically I like to wrap the Source with a BUILD function to remove any spaces (e.g. BUILD(NURSE_UNIT)).
- **Condition** is used when you want to control when the field prints. Using a simple WHERE statement, your field will only display when the condition is true.

The remainder of the fields are used to control how your text is displayed. You can change properties such as font, color, underline, wrapping, etc.

To continue our example, set the following properties for the field you created.

Name: NurseUnit
Source: BUILD(NURSE_UNIT)

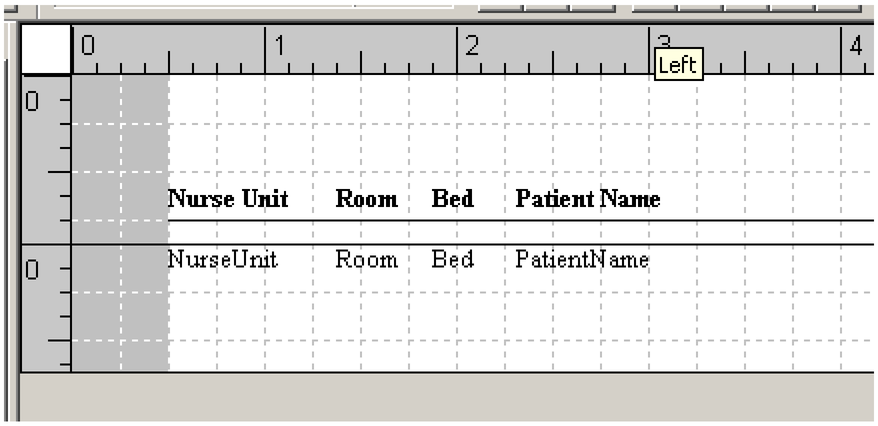
Once you have finished the first field, create three other fields and name them as follows:

Name: Room
Source: BUILD(ROOM)

Name: Bed
Source: BUILD(BED)

Name: PatientName
Source: BUILD(PATIENT_NAME)

Once the fields are created, you can re-size and position them so they line up with the titles. Your two report sections should look something like the following:



Nurse Unit	Room	Bed	Patient Name
NurseUnit	Room	Bed	PatientName

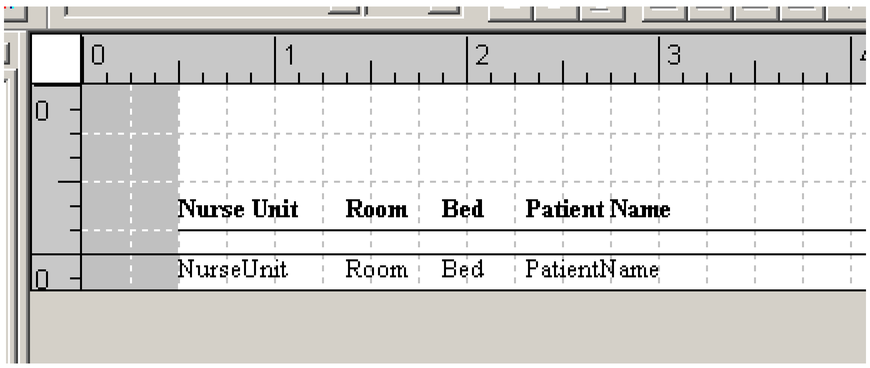
Now let's take a moment and discuss our Body section. When we go to print the body section, we will likely want to have the details print one after another rather than with a large three quarter inch gap between them. By default, the new section we have created is ¾" tall. This is much too large of a gap between detail lines on a report.

You can resize the section (smaller or larger) by performing the following steps:

1. Hover your mouse over the solid black line that defines the bottom of your section. The mouse pointer will change to a pair of up and down arrows.
2. Hold down the left mouse button and drag the mouse up or down to change the size of the section. Release the mouse when the section is the size you want.

If you drag your section as small as it will go, you will notice that it stops at a certain point. This is the boundary of the lowest place item in the section. The text fields we have created are actually ¼" tall and unless we change the size of those items, the smallest our section can ever be is ¼" tall.

To shorten the section even further, re-size your four fields and finish up by making your section smaller. Your two sections should now appear as follows:



Nurse Unit Room Bed Patient Name

NurseUnit Room Bed PatientName

You are now almost done writing your first Layout Builder report. The final step is to put the code in your script that will execute the Layout Builder sections you just created. Close Layout builder by clicking on the X in the top right corner of the screen that is on the same line as the file menu.

You should now be back to your source code. To make your layout builder code work, there are only a few small pieces of code that need to be added.

For this example, we need to add a prompt. This can be done by manually adding a prompt or using the prompt builder. If your report is to be executed from Explorer Menu, a prompt is required and the first prompt must be the output device. For our example we are going to use a single prompt. Place the following code right after the CREATE PROGRAM line in your script.

```
PROMPT “Output to File/Printer/Mine”=”MINE”
WITH OUTDEV
```

Next, you need to add the code written by Layout Builder to your report. The following lines of code need to be added somewhere after your prompt statement but before the SELECT statement that is going to use the Layout Builder. In this example, you can put this code immediately after the PROMPT statement.

```
EXECUTE REPORTRTL
```

```
%INCLUDE TEST_yourinitials.DVL
```

```
CALL InitializeReport(0)
```

The second line of text shown above will be different in every program you write. As you work in Layout Builder, a file is created that matches your program name and it contains all of the code needed for your report.

The InitializeReport(0) function is actually a function defined in the DVL file created by Layout Builder. When executed it performs all the steps necessary to initialize variables needed in your report. However, there is one variable that you need to manually initialize to set the way you need. That variable is the variable used to determine if you need to force a page break. The code to set this variable is as follows and can be placed immediately after the CALL InitializeReport(0) line.

If your report is a portrait report, use the following:

```
      SET _fEndDetail = RptReport->m_pageHeight – RptReport->m_marginBottom
```

If your report is a landscape report, use the following:

```
      SET _fEndDetail = RptReport->m_pageWidth – RptReport->m_marginRight
```

Next, we want to start calling the code in our Layout to print our header and body sections. To do this, add the following code at the bottom of our SELECT statement.

```
HEAD REPORT
      X = Header(0)
DETAIL
      IF (_YOffset + Body(1) > _fEndDetail)
            CALL PageBreak(0)
            X = Header(0)
      ENDIF

      X = Body(0)
WITH NOCOUNTER
```

In the code above, we start by printing our Header section at the top of the page. All of the code necessary for printing the Header section is in a function called Header(). To execute the section, we simply call the function name with a parameter of 0.

In the detail of our report, we want to print the Body section. This section can print many times over (one time for every line of detail in our report). Printing past the bottom of the page is likely in our detail so we must test to see if the next print of our Body section is greater than the bottom of the page. By passing the number one as a parameter to the Body section, we are returned a value the represents where the next print of the Body section will be on the page. If it is greater than _fEndDetail, we issue a PageBreak and print our page Header section again.

Our final step in the report detail is to print our Body section. This is done by executing the Body function with a parameter of 0.

Now, we are at the final step in our code. Outside of our SQL statement, we need to issue the FinalizeReport function. To do this, add the following line just before the END GO statement:

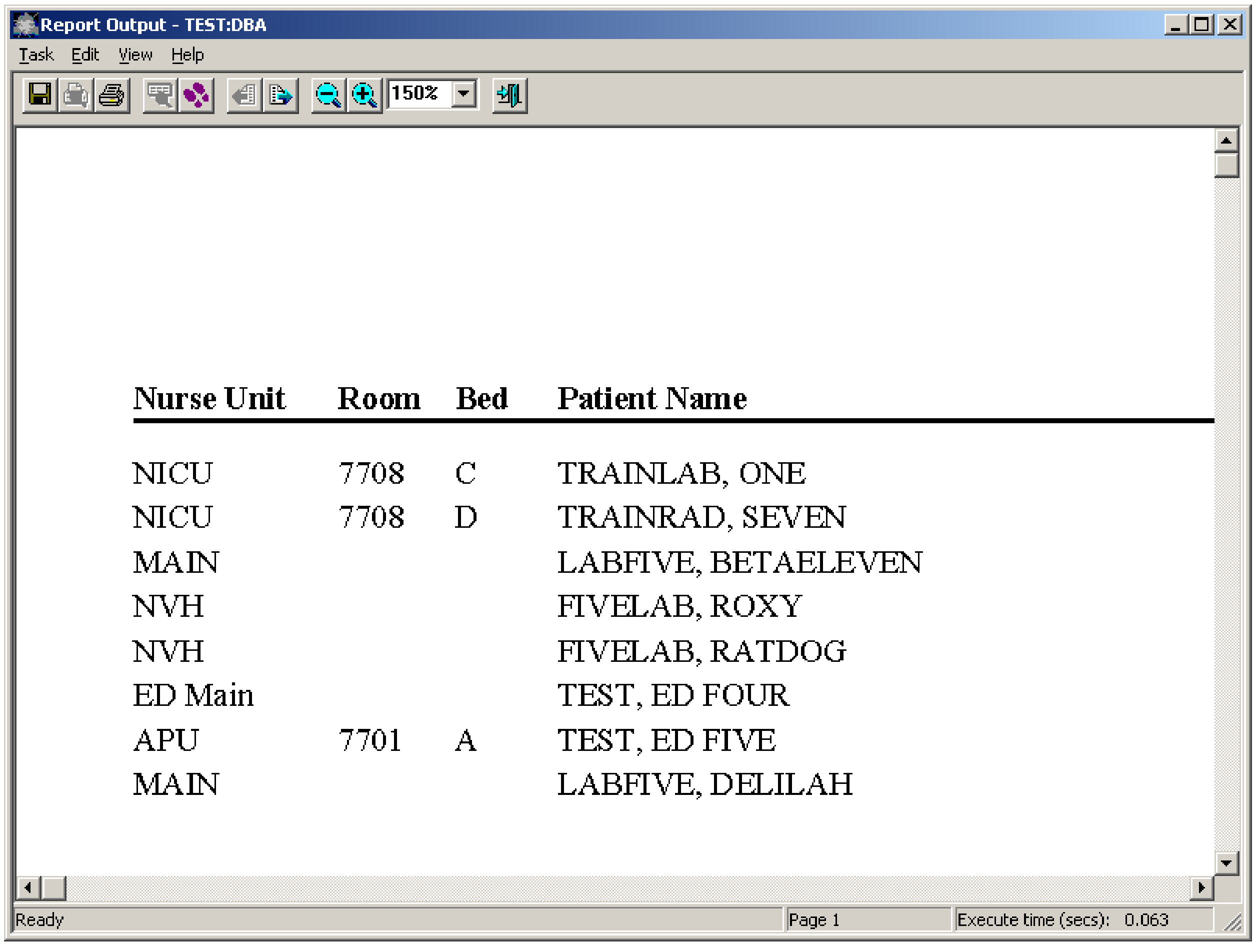
```
CALL FinalizeReport($OUTDEV)
```

This line simply tells the Layout code that the report is finished and the output should be directed to the device indicated in the $OUTDEV variable.

Your final report code should be as follows:

```
DROP PROGRAM TEST:DBA GO
CREATE PROGRAM TEST:DBA

PROMPT "Output to File/Printer/Mine"="MINE"
WITH OUTDEV

EXECUTE REPORTRTL

%INCLUDE TEST.DVL

CALL InitializeReport(0)

IF (RptReport->M_Orientation = 0)
      SET _fEndDetail = RptReport->m_pageHeight - RptReport->m_marginBottom
ELSE
      SET _fEndDetail = RptReport->m_pageWidth - RptReport->m_marginRight
ENDIF

SELECT INTO "NL:"
      NURSE_UNIT        = UAR_GET_CODE_DISPLAY(E.LOC_NURSE_UNIT_CD),
      ROOM              = UAR_GET_CODE_DISPLAY(E.LOC_ROOM_CD),
      BED               = UAR_GET_CODE_DISPLAY(E.LOC_BED_CD),
      PATIENT_NAME      = P.NAME_FULL_FORMATTED
FROM  ENCOUNTER         E,
            PERSON            P
PLAN  E
      WHERE E.REG_DT_TM > CNVTDATETIME(CURDATE-1, CURTIME3)
      AND   E.ACTIVE_IND = 1
JOIN  P
      WHERE P.PERSON_ID = E.PERSON_ID
HEAD REPORT
      X = Header(0)
DETAIL
      IF (_YOffset + Body(1) > _fEndDetail)
            CALL PageBreak(0)
            X = Header(0)
      ENDIF

      X = Body(0)
WITH NOCOUNTER

CALL FinalizeReport($OUTDEV)

END GO
```

When you execute your report from DiscernVisualDeveloper, you will be asked for a printer name. Leave the value as MINE and click the Execute button to view your report on screen. The result should appear similar to the following output:



```
Nurse Unit    Room   Bed   Patient Name
---------------------------------------------------
NICU          7708   C     TRAINLAB, ONE
NICU          7708   D     TRAINRAD, SEVEN
MAIN                       LABFIVE, BETAELEVEN
NVH                        FIVELAB, ROXY
NVH                        FIVELAB, RATDOG
ED Main                    TEST, ED FOUR
APU           7701   A     TEST, ED FIVE
MAIN                       LABFIVE, DELILAH
```

Word Wrapping

Layout Builder has come a long way from the traditional manual methods of wrapping text in CCL. Instead of calculating font sizes manually, you can now set a few simple parameters and execute a small section of code to offer your end users flawless word wrapping.

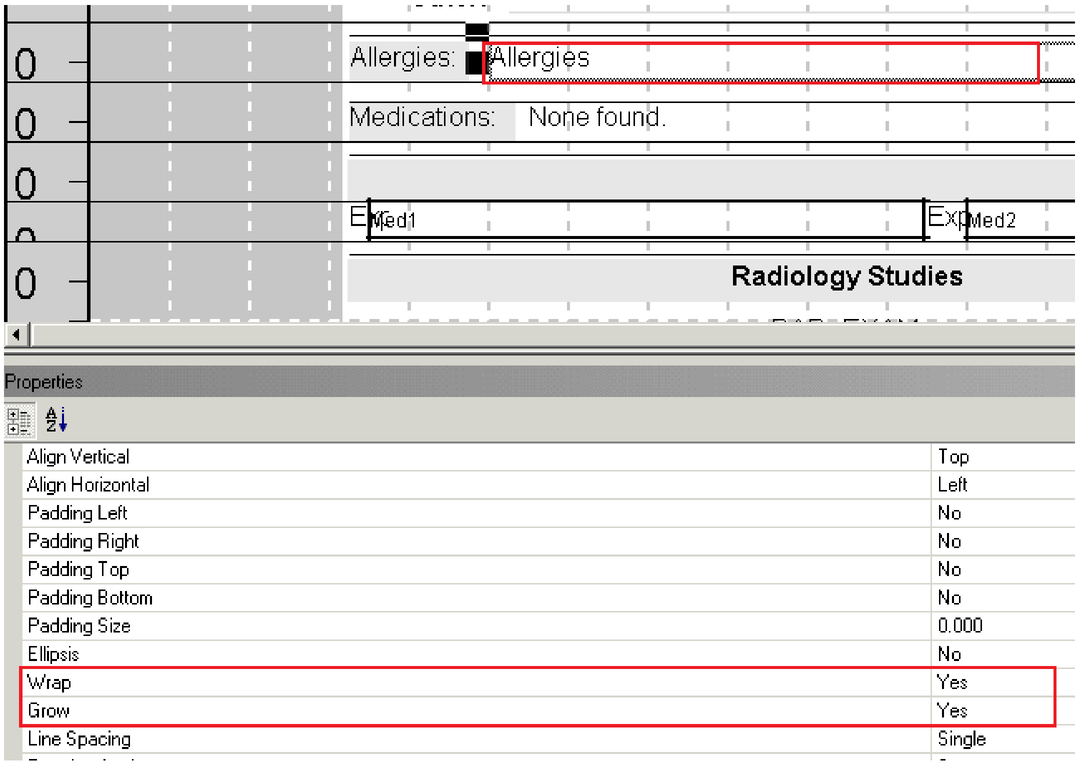
The following steps will describe how to create a Layout Builder section that will not only wrap text, but grow the section to fit the content.

First, create a new section in Layout Builder. For my example, I have called my section ***Allergies***.

Your second step is to add a text field in the ***Allergies*** section. This will contain the data you wish to wrap. ***Note: You can also add other fields in the section (see the Allergies: label in the screen shot below).***

To just offer wrapping without growing the section, you can size the field to the height you want and simply set the ***Wrap*** property of the field to ***Yes***. If the text is longer than the width of the field, it will wrap to the height defined for your text field. Any text printed beyond the regions of the height of the field will be lost.

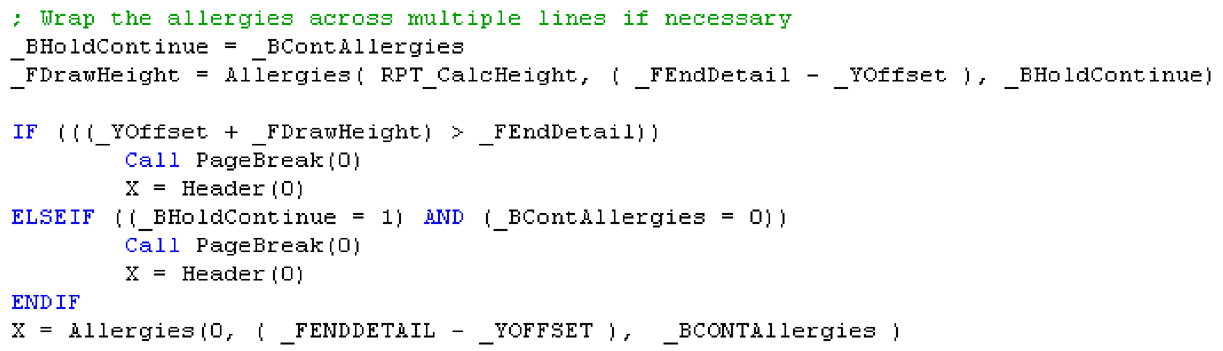
However, in many cases, you will want the report to print the entire contents of your text field regardless of how much data is being passed to the field. To do this, you will also need to set the ***Grow*** property to ***Yes*** in addition to adding some code to handle any page breaks.



Align Vertical	Top
Align Horizontal	Left
Padding Left	No
Padding Right	No
Padding Top	No
Padding Bottom	No
Padding Size	0.000
Ellipsis	No
Wrap	Yes
Grow	Yes
Line Spacing	Single

Once you have set your properties, go back to your source code and enter the following code (Change the word Allergies to whatever your section name is called).

OLD NOT SURE IF IT WORKS



```
; Wrap the allergies across multiple lines if necessary
_BHoldContinue = _BContAllergies
_FDrawHeight = Allergies( RPT_CalcHeight, ( _FEndDetail - _YOffset ), _BHoldContinue)

IF (((_YOffset + _FDrawHeight) > _FEndDetail))
        Call PageBreak(0)
        X = Header(0)
ELSEIF ((_BHoldContinue = 1) AND (_BContAllergies = 0))
        Call PageBreak(0)
        X = Header(0)
ENDIF
X = Allergies(0, ( _FENDDETAIL - _YOFFSET ),  _BCONTAllergies )
```

NEW METHOD

```
_BHoldContinue = _BContAllergies
_FDrawHeight = Allergies(RPT_CalcHeight, (_FEndDetail-_YOffset), _BHoldContinue)

IF ((_YOffset + _FDrawHeight > _FEndDetail)
             OR (_BHoldContinue = 1 AND _BContAllergies = 1))
    Call PageBreak(0)
    X = Header(0)
ENDIF

X = Allergies(0, (_FEndDetail - _YOffset), _BContAllergies)

WHILE (_BContAllergies > 0)
    CALL PageBreak(0)
    X = Header(0)
    X = Allergies(0, (_FEndDetail - _YOffset), _BContAllergies)
ENDWHILE
```

This code will not only grow the wrapped field, it will also cause a page break and continue printing on the next page if there is still content to be printed.

Please note that ***Header(0)*** is the name of the page header section defined for this sample report. Your header section may be named something different or not exist at all.

Appendix

A1.1 – The TRANSLATE command

Often you may know the compiled object name of a script but not have the source code available. The majority of Cerner written reports are provided without source code.

You can de-compile CCL reports with the translate command.

From the back-end CCL prompt, issue the translate command along with the compiled object name to have the source displayed on screen.

For example:

```
TRANSLATE DCP_RPT_PVPATLIST GO
```

Often you will want to save the translated source as a file. To do so, simply include the INTO filename parameter before the object name.

```
TRANSLATE INTO sourcefilename DCP_RPT_PVPATLIST GO
```

The TRANSLATE command will send the output file to the CCLUSERDIR directory with a .CCL extension

Like de-compilers in other languages, the CCL TRANSLATE command does lose commenting and text spacing in the source code will be affected. The TRANSLATE command should only be used when source code is unavailable.

A1.2 – The CCLPROT command

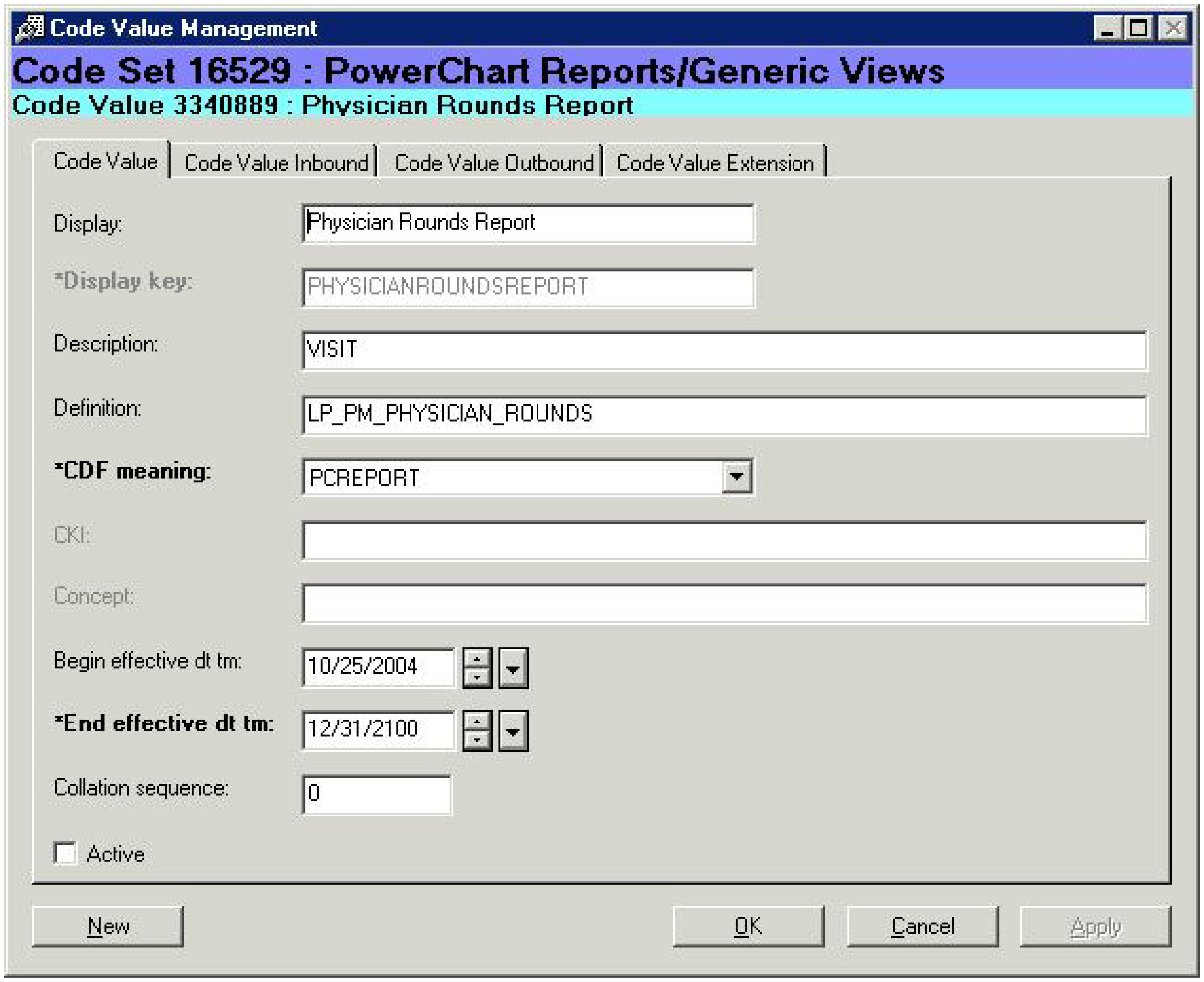
The CCLPROT command can be used to help determine how a CCL script was compiled (which security level) and where the source code was located at compile time. Simply type in CCLPROT GO at the CCL command line and fill in the prompts. (Hint: Object Name and Display Source Filename are the only fields you need to change)

A 2.1 - Customizing the Patient List Print Button in PowerChart

Changing the functionality of the Patient List Print button in Powerchart is as simple as developing your CCL report and adding a few pieces of code. Additionally you need to add an entry to code set 16529 and modify the preferences in prefmaint.exe for the position you would like to use the custom report in.

First, we will add the new entry in code set 16529. To do this, launch corecodebuilder.exe and search for code set 16529. Click the Add button to add a new code value. In the next screen that appears, enter the following information.

Display:	A descriptive name for your report
Description:	VISIT
Definition:	The compiled object name of your report (minus the .prg)
CDF Meaning:	PCREPORT


Code Value Management

Code Set 16529 : PowerChart Reports/Generic Views

Code Value 3340889 : Physician Rounds Report

Display:	Physician Rounds Report
*Display key:	PHYSICIANROUNDSREPORT
Description:	VISIT
Definition:	LP_PM_PHYSICIAN_ROUNDS
*CDF meaning:	PCREPORT
CKI:	
Concept:	
Begin effective dt tm:	10/25/2004
*End effective dt tm:	12/31/2100
Collation sequence:	0
Active	☐

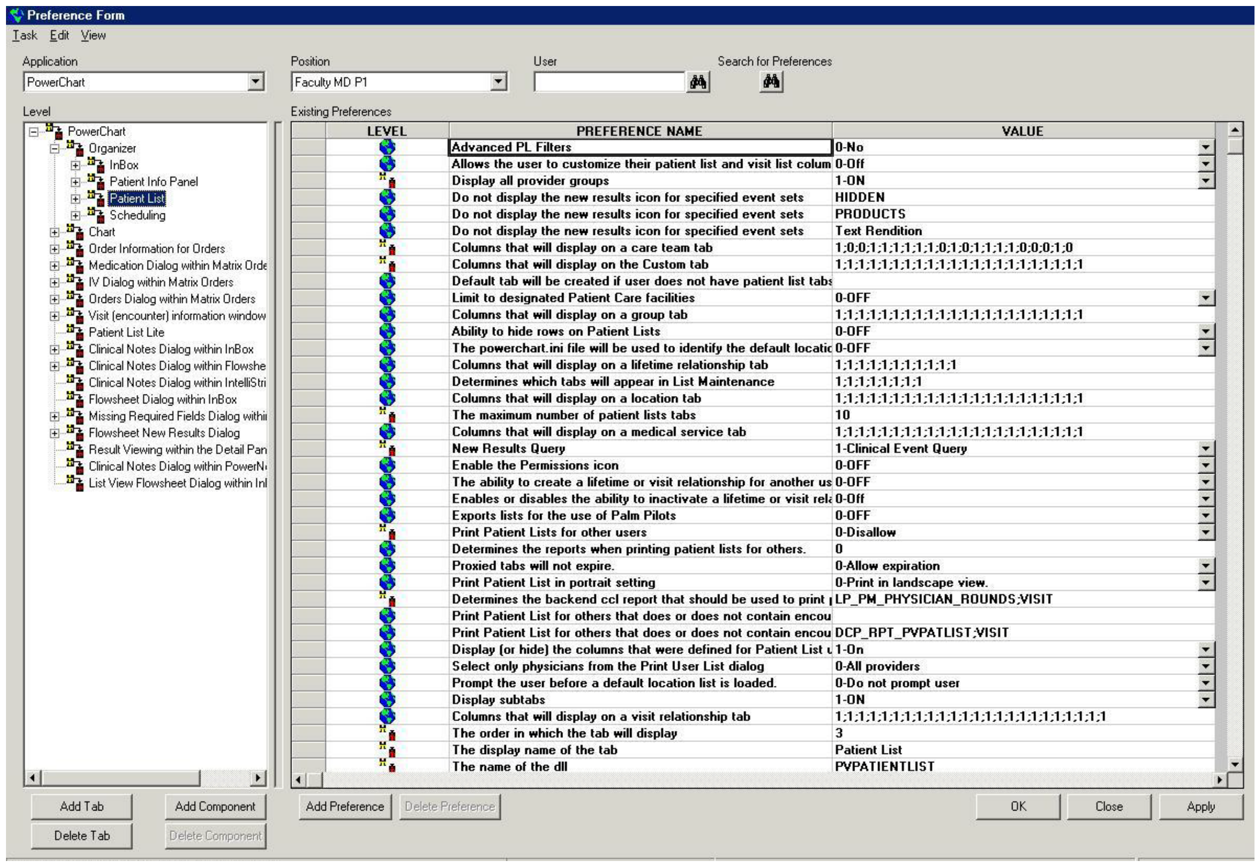
The next step is to add the report to the user preferences for the position you wish to print the report for. This is done in the prefmaint.exe tool. Run prefmaint.exe, select the Application of PowerChart and choose the position you wish to apply the report to.

*** Warning ** prefmaint.exe tends not to work on computer systems running dual monitors. If you are running dual monitors, switch to a single monitor view or use a different PC when making changes in prefmaint.exe*

The setting you are looking for is in the PowerChart->Organizer->Patient List folder. It is called "Determines the backend ccl report that should be used to print patient lists". If you are viewing by PVC Name, the name is "PTLIST_REPORT".

If the field is not present on your screen, you may have to click the Add Preference button and add the preference.

In the value for the preference, enter the object name of your report followed by ;VISIT. In my example, the entry is "LP_PM_PHYSICIAN_ROUNDS;VISIT".



Application: PowerChart

Position: Faculty MD P1

User

Search for Preferences

Level

- PowerChart
  - Organizer
    - InBox
    - Patient Info Panel
    - Patient List
    - Scheduling
  - Chart
  - Order Information for Orders
  - Medication Dialog within Matrix Orde
  - IV Dialog within Matrix Orders
  - Orders Dialog within Matrix Orders
  - Visit (encounter) information window
  - Patient List Lite
  - Clinical Notes Dialog within InBox
  - Clinical Notes Dialog within Flowshe
  - Clinical Notes Dialog within IntelliStri
  - Flowsheet Dialog within InBox
  - Missing Required Fields Dialog withi
  - Flowsheet New Results Dialog
  - Result Viewing within the Detail Pan
  - Clinical Notes Dialog within PowerN
  - List View Flowsheet Dialog within Inl

Existing Preferences

LEVEL	PREFERENCE NAME	VALUE
🌐	Advanced PL Filters	0-No
🌐	Allows the user to customize their patient list and visit list colum	0-Off
▪	Display all provider groups	1-ON
🌐	Do not display the new results icon for specified event sets	HIDDEN
🌐	Do not display the new results icon for specified event sets	PRODUCTS
🌐	Do not display the new results icon for specified event sets	Text Rendition
▪	Columns that will display on a care team tab	1;0;0;1;1;1;1;1;1;0;1;0;1;1;1;1;0;0;0;1;0
▪	Columns that will display on the Custom tab	1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1
🌐	Default tab will be created if user does not have patient list tabs	
🌐	Limit to designated Patient Care facilities	0-OFF
🌐	Columns that will display on a group tab	1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1
🌐	Ability to hide rows on Patient Lists	0-OFF
🌐	The powerchart.ini file will be used to identify the default locatic	0-OFF
🌐	Columns that will display on a lifetime relationship tab	1;1;1;1;1;1;1;1;1;1
🌐	Determines which tabs will appear in List Maintenance	1;1;1;1;1;1;1;1
🌐	Columns that will display on a location tab	1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1
▪	The maximum number of patient lists tabs	10
🌐	Columns that will display on a medical service tab	1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1
▪	New Results Query	1-Clinical Event Query
🌐	Enable the Permissions icon	0-OFF
🌐	The ability to create a lifetime or visit relationship for another us	0-OFF
🌐	Enables or disables the ability to inactivate a lifetime or visit rel	0-Off
🌐	Exports lists for the use of Palm Pilots	0-OFF
▪	Print Patient Lists for other users	0-Disallow
🌐	Determines the reports when printing patient lists for others.	0
🌐	Proxied tabs will not expire.	0-Allow expiration
🌐	Print Patient List in portrait setting	0-Print in landscape view.
▪	Determines the backend ccl report that should be used to print	LP_PM_PHYSICIAN_ROUNDS;VISIT
🌐	Print Patient List for others that does or does not contain encou	
🌐	Print Patient List for others that does or does not contain encou	DCP_RPT_PVPATLIST;VISIT
🌐	Display (or hide) the columns that were defined for Patient List u	1-On
🌐	Select only physicians from the Print User List dialog	0-All providers
🌐	Prompt the user before a default location list is loaded.	0-Do not prompt user
🌐	Display subtabs	1-ON
🌐	Columns that will display on a visit relationship tab	1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1
▪	The order in which the tab will display	3
▪	The display name of the tab	Patient List
▪	The name of the dll	PVPATIENTLIST

The final step is to ensure that your report is setup to handle the record structure passed from PowerChart.

The following structure contains the basic information needed for generating an encounter based rounds report:

```
RECORD REQUEST (
     1 OUTPUT_DEVICE       = vc
     1 VISIT_CNT           = i4
     1 VISIT[*]
          2 ENCNTR_ID      = f8
)
```

If you are using layout builder, you will need to have your FinalizeReport statement refer to the output device in the request structure. For straight SELECT INTO (outdev) type reports, pass the output_device in the statement.

CALL FinalizeReport(REQUEST->OUTPUT_DEVICE)

Or

SELECT INTO VALUE(REQUEST->OUTPUT_DEVICE)

The OUTPUT_DEVICE represents the spool location of the output file generate by your report. Outputting your report to the value in this variable will not print the report but rather generate a file. To print the report you will need to issue the following command.

```
SET REPLY->TEXT = VALUE(REQUEST->OUTPUT_DEVICE)
```

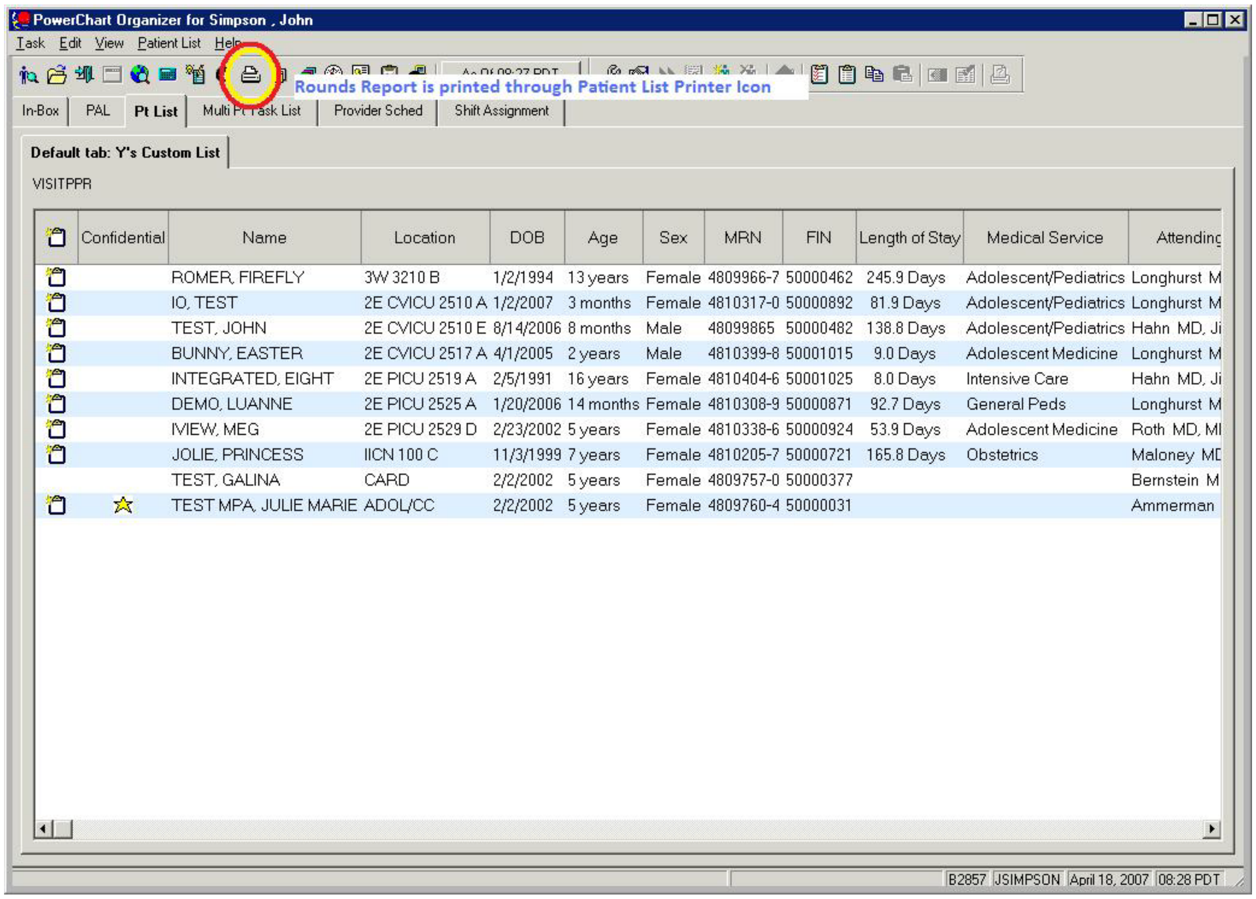
However, before you get to actually printing your report, you will need to collect your report data. The starting point is to PLAN off the REQUEST->VISIT structure. This structure contains all of the encounters listed in the patient list that is on screen when the print icon is clicked in PowerChart.

The simplified code to do this is:

```
SELECT
FROM  (DUMMYT          D WITH SEQ=VALUE(REQUEST->VISIT_CNT)),
      ENCOUNTER        E,
      ...more tables...
PLAN  D
JOIN  E
      WHERE E.ENCNTR_ID = REQUEST->VISIT[D.SEQ].ENCNTR_ID
JOIN  ...other table...
```

The content of your report is entirely up to you (and the specifications you have), but putting this type of report together is no more difficult than any other report. In fact, many of the performance issues you may face in the data collection of other reports are limited since your encntr_id's are pre-established for you.

When you are done coding your report, adding the new code value to code set 16529 and modifying the user preferences, your end users should be able to print from the printer icon in the Patient List in PowerChart.


PowerChart Organizer for Simpson , John
Task Edit View Patient List Help
In-Box PAL Pt List Multi Pt Task List Provider Sched Shift Assignment
Default tab: Y's Custom List
VISITPPR
B2857 JSIMPSON April 18, 2007 08:28 PDT

Rounds Report is printed through Patient List Printer Icon

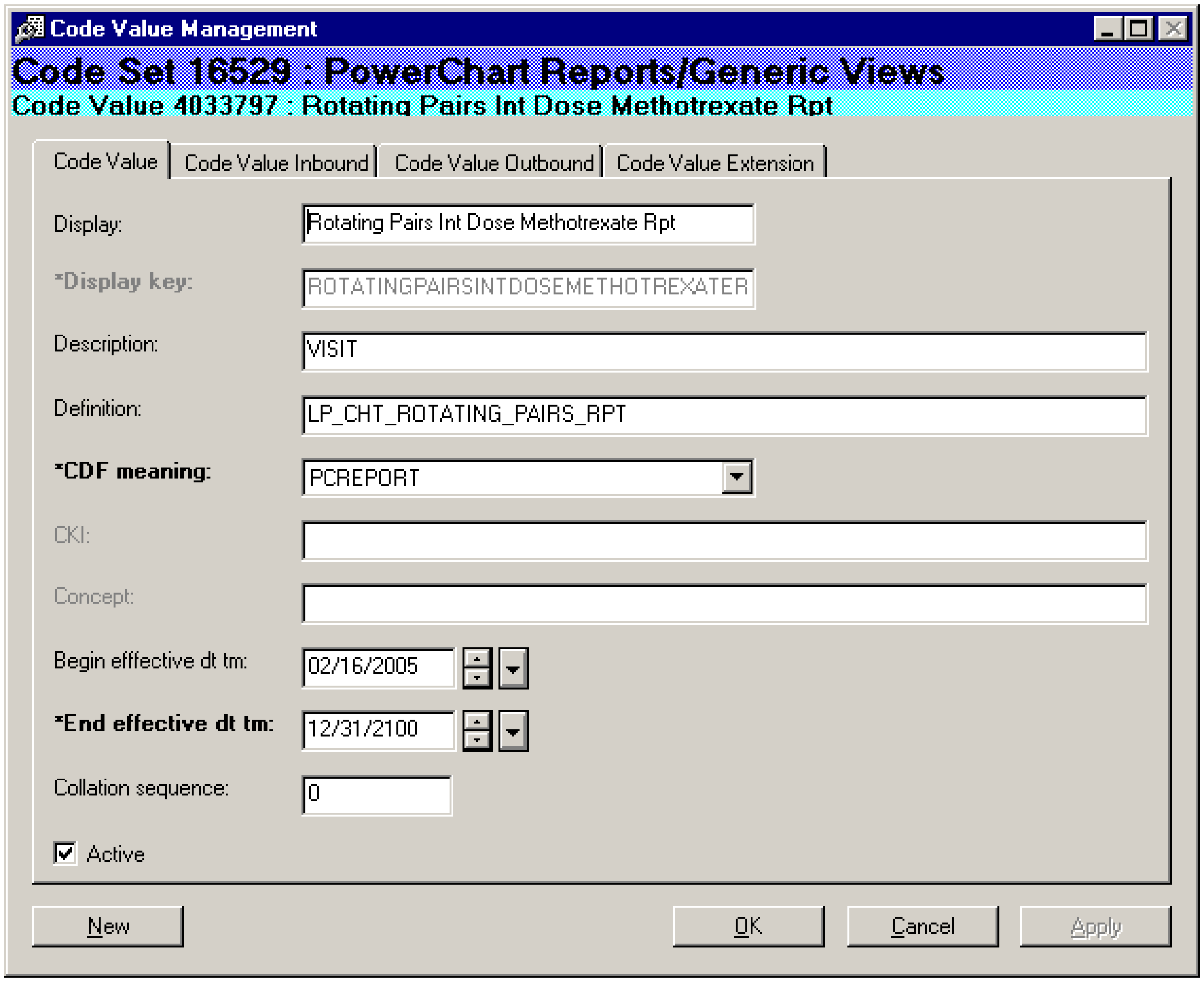
📋	Confidential	Name	Location	DOB	Age	Sex	MRN	FIN	Length of Stay	Medical Service	Attending
📋		ROMER, FIREFLY	3W 3210 B	1/2/1994	13 years	Female	4809966-7	50000462	245.9 Days	Adolescent/Pediatrics	Longhurst M
📋		IO, TEST	2E CVICU 2510 A	1/2/2007	3 months	Female	4810317-0	50000892	81.9 Days	Adolescent/Pediatrics	Longhurst M
📋		TEST, JOHN	2E CVICU 2510 E	8/14/2006	8 months	Male	48099865	50000482	138.8 Days	Adolescent/Pediatrics	Hahn MD, Ji
📋		BUNNY, EASTER	2E CVICU 2517 A	4/1/2005	2 years	Male	4810399-8	50001015	9.0 Days	Adolescent Medicine	Longhurst M
📋		INTEGRATED, EIGHT	2E PICU 2519 A	2/5/1991	16 years	Female	4810404-6	50001025	8.0 Days	Intensive Care	Hahn MD, Ji
📋		DEMO, LUANNE	2E PICU 2525 A	1/20/2006	14 months	Female	4810308-9	50000871	92.7 Days	General Peds	Longhurst M
📋		IVIEW, MEG	2E PICU 2529 D	2/23/2002	5 years	Female	4810338-6	50000924	53.9 Days	Adolescent Medicine	Roth MD, MI
📋		JOLIE, PRINCESS	IICN 100 C	11/3/1999	7 years	Female	4810205-7	50000721	165.8 Days	Obstetrics	Maloney MD
		TEST, GALINA	CARD	2/2/2002	5 years	Female	4809757-0	50000377			Bernstein M
📋	☆	TEST MPA, JULIE MARIE	ADOL/CC	2/2/2002	5 years	Female	4809760-4	50000031			Ammerman

Adding a report to the PowerChart reports menu

Adding a report to the PowerChart reports menu involves entering a new record into code set 16529 and some simple modifications to your CCL code. Once complete, your report will appear in the PowerChart reports menu.

First, we will add the new entry in code set 16529. To do this, launch corecodebuilder.exe and search for code set 16529. Click the Add button to add a new code value. In the next screen that appears, enter the following information.

Display:	A descriptive name for your report
Description:	VISIT
Definition:	The compiled object name of your report (minus the .prg)
CDF Meaning:	PCREPORT


Code Value Management

Code Set 16529 : PowerChart Reports/Generic Views

Code Value 4033797 : Rotating Pairs Int Dose Methotrexate Rpt

Display:	Rotating Pairs Int Dose Methotrexate Rpt
*Display key:	ROTATINGPAIRSINTDOSEMETHOTREXATER
Description:	VISIT
Definition:	LP_CHT_ROTATING_PAIRS_RPT
*CDF meaning:	PCREPORT
CKI:	
Concept:	
Begin efffective dt tm:	02/16/2005
*End effective dt tm:	12/31/2100
Collation sequence:	0

☑ Active

The final step is to ensure that your report is setup to handle the record structure passed from PowerChart.

The following structure contains the basic information needed for generating a report:

```
RECORD REQUEST (
     1 NV_CNT                  = i4
     1 NV[2]
          2 PVC_NAME           = c10
          2 PVC_VALUE          = c16
     1 VISIT[1]
          2 ENCNTR_ID          = f8
     1 OUTPUT_DEVICE           = c50
)
```

When run from the Powerchart Reports menu, an Output Destination, ENCNTR_ID, and to and from dates are passed to your CCL script. To handle these parameters, copy the following block of code into your CCL script:

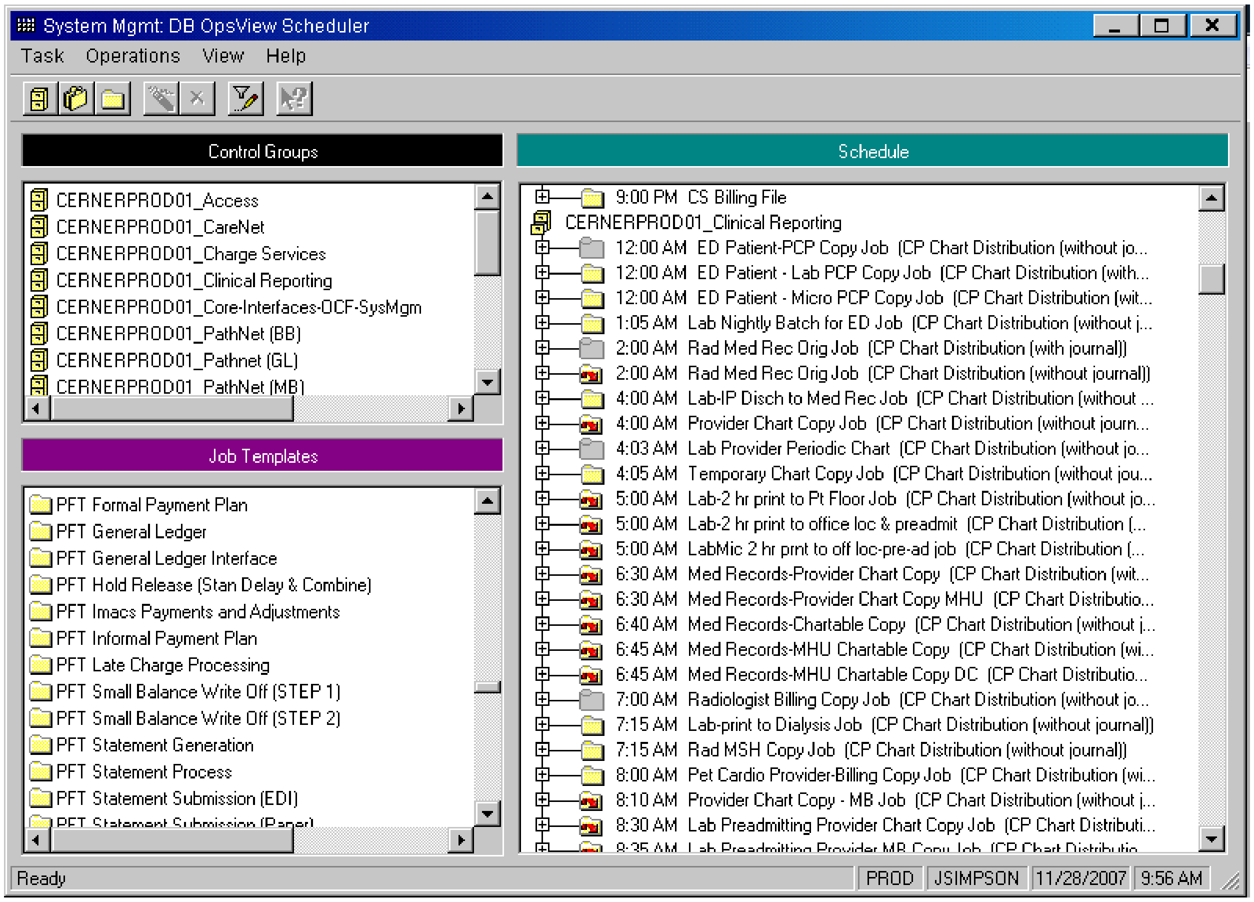
```
; Create copied variables of request structure information to later support batch or explorer
menu printing
SET nENCNTR_ID = REQUEST->VISIT[1].ENCNTR_ID
SET OUTDEV = REQUEST->OUTPUT_DEVICE

; Converts a 16 character PVC_VALUE to a Date/Time value
SUBROUTINE PVC_TO_DATE(PVC_VALUE)
        RETURN
(CNVTDATETIME(CNVTDATE2(SUBSTRING(1,8,PVC_VALUE),"YYYYMMDD"),CNVTINT(SUBSTRING(9,4,PVC_VALUE))))
END

; Collect the dates entered in PowerChart
SET nBEG_DT_TM = CNVTDATETIME(CURDATE, CURTIME3)
SET nEND_DT_TM = CNVTDATETIME(CURDATE, CURTIME3)

FOR (nLOOP = 1 TO REQUEST->NV_CNT)
        CASE (REQUEST->NV[nLOOP]->PVC_NAME)
                OF "BEG_DT_TM": SET nBEG_DT_TM = PVC_TO_DATE(REQUEST->NV[nLOOP]->PVC_VALUE)
                OF "END_DT_TM": SET nEND_DT_TM = PVC_TO_DATE(REQUEST->NV[nLOOP]->PVC_VALUE)
        ENDCASE
ENDFOR
```

Once the code above is inserted into your script, the following variables will be assigned for use in your report:

OUTDEV	- The output destination (in this case the printer queue name)
nENCNTR_ID	- Contains the ENCNTR_ID you are reporting on
nBEG_DT_TM	- A date/time format begin date/time
nEND_DT_TM	- A date/time format end date/time

A2.3 - Adding a Report to Operations

Batches are added to Operations through the OPSViewScheduler.exe application. (See screenshot below)



CERNERPROD01_Access
CERNERPROD01_CareNet
CERNERPROD01_Charge Services
CERNERPROD01_Clinical Reporting
CERNERPROD01_Core-Interfaces-OCF-SysMgm
CERNERPROD01_PathNet (BB)
CERNERPROD01_Pathnet (GL)
CERNERPROD01_PathNet (MB)

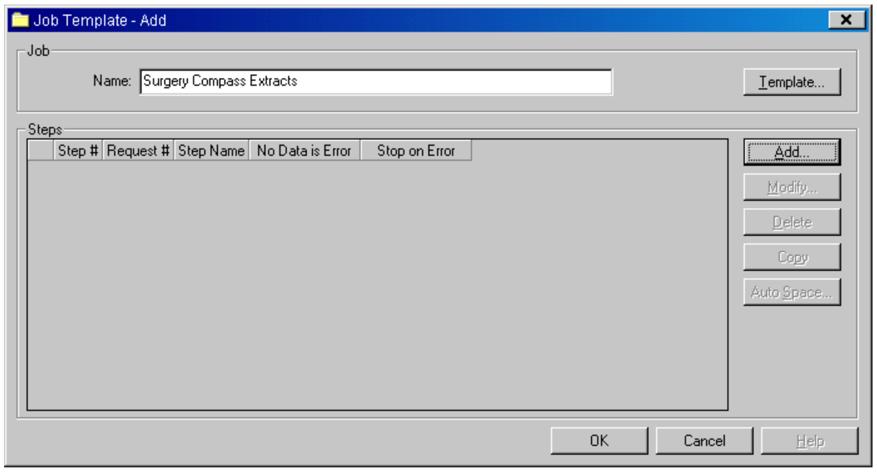
PFT Formal Payment Plan
PFT General Ledger
PFT General Ledger Interface
PFT Hold Release (Stan Delay & Combine)
PFT Imacs Payments and Adjustments
PFT Informal Payment Plan
PFT Late Charge Processing
PFT Small Balance Write Off (STEP 1)
PFT Small Balance Write Off (STEP 2)
PFT Statement Generation
PFT Statement Process
PFT Statement Submission (EDI)
PFT Statement Submission (Paper)

9:00 PM CS Billing File
CERNERPROD01_Clinical Reporting
12:00 AM ED Patient-PCP Copy Job (CP Chart Distribution (without jo...
12:00 AM ED Patient - Lab PCP Copy Job (CP Chart Distribution (with...
12:00 AM ED Patient - Micro PCP Copy Job (CP Chart Distribution (wit...
1:05 AM Lab Nightly Batch for ED Job (CP Chart Distribution (without j...
2:00 AM Rad Med Rec Orig Job (CP Chart Distribution (with journal))
2:00 AM Rad Med Rec Orig Job (CP Chart Distribution (without journal))
4:00 AM Lab-IP Disch to Med Rec Job (CP Chart Distribution (without ...
4:00 AM Provider Chart Copy Job (CP Chart Distribution (without journ...
4:03 AM Lab Provider Periodic Chart (CP Chart Distribution (without jo...
4:05 AM Temporary Chart Copy Job (CP Chart Distribution (without jou...
5:00 AM Lab-2 hr print to Pt Floor Job (CP Chart Distribution (without jo...
5:00 AM Lab-2 hr print to office loc & preadmit (CP Chart Distribution (...
5:00 AM LabMic 2 hr prnt to off loc-pre-ad job (CP Chart Distribution (...
6:30 AM Med Records-Provider Chart Copy (CP Chart Distribution (wit...
6:30 AM Med Records-Provider Chart Copy MHU (CP Chart Distributio...
6:40 AM Med Records-Chartable Copy (CP Chart Distribution (without j...
6:45 AM Med Records-MHU Chartable Copy (CP Chart Distribution (wi...
6:45 AM Med Records-MHU Chartable Copy DC (CP Chart Distributio...
7:00 AM Radiologist Billing Copy Job (CP Chart Distribution (without jo...
7:15 AM Lab-print to Dialysis Job (CP Chart Distribution (without journal))
7:15 AM Rad MSH Copy Job (CP Chart Distribution (without journal))
8:00 AM Pet Cardio Provider-Billing Copy Job (CP Chart Distribution (wi...
8:10 AM Provider Chart Copy - MB Job (CP Chart Distribution (without j...
8:30 AM Lab Preadmitting Provider Chart Copy Job (CP Chart Distributi...
8:35 AM Lab Preadmitting Provider MB Copy Job (CP Chart Distributio...

Ready PROD JSIMPSON 11/28/2007 9:56 AM

First, you need to create a Job Template. This is done by clicking the Job Template – Add icon. Once clicked, you will see the screen in which you will add your Job Template.

Fill in the name property and click the Add button to add the job steps.



Job Template - Add

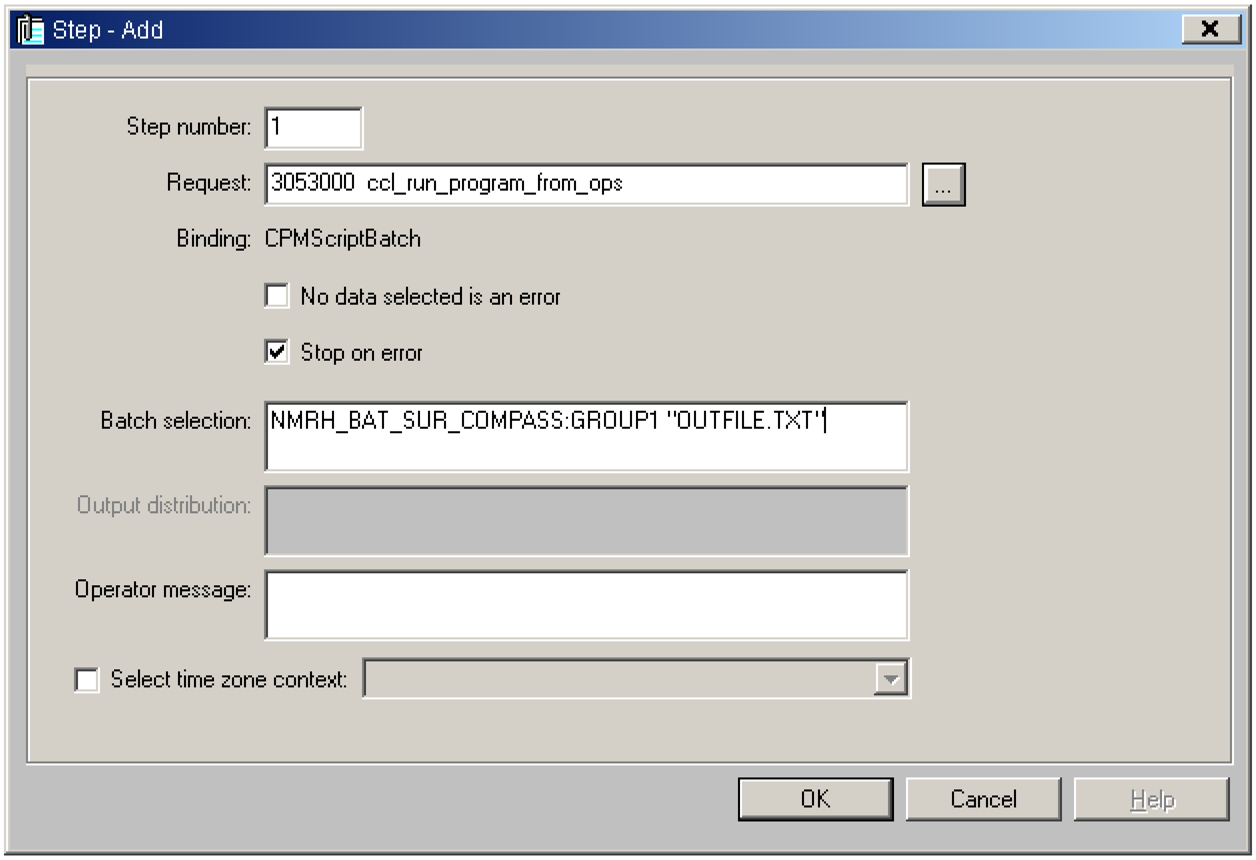
Job

Name: Surgery Compass Extracts

Step #	Request #	Step Name	No Data is Error	Stop on Error

Complete the step fields by filling in the following and click the Ok button:

Step number: 1
Request: 3053000 (ccl_run_program_from_ops will automatically fill in)
Batch selection: your batch script name followed by the security group and any prompts required by the CCL script..



Step - Add

Step number: 1

Request: 3053000 ccl_run_program_from_ops

Binding: CPMScriptBatch

☐ No data selected is an error

☑ Stop on error

Batch selection: NMRH_BAT_SUR_COMPASS:GROUP1 "OUTFILE.TXT"

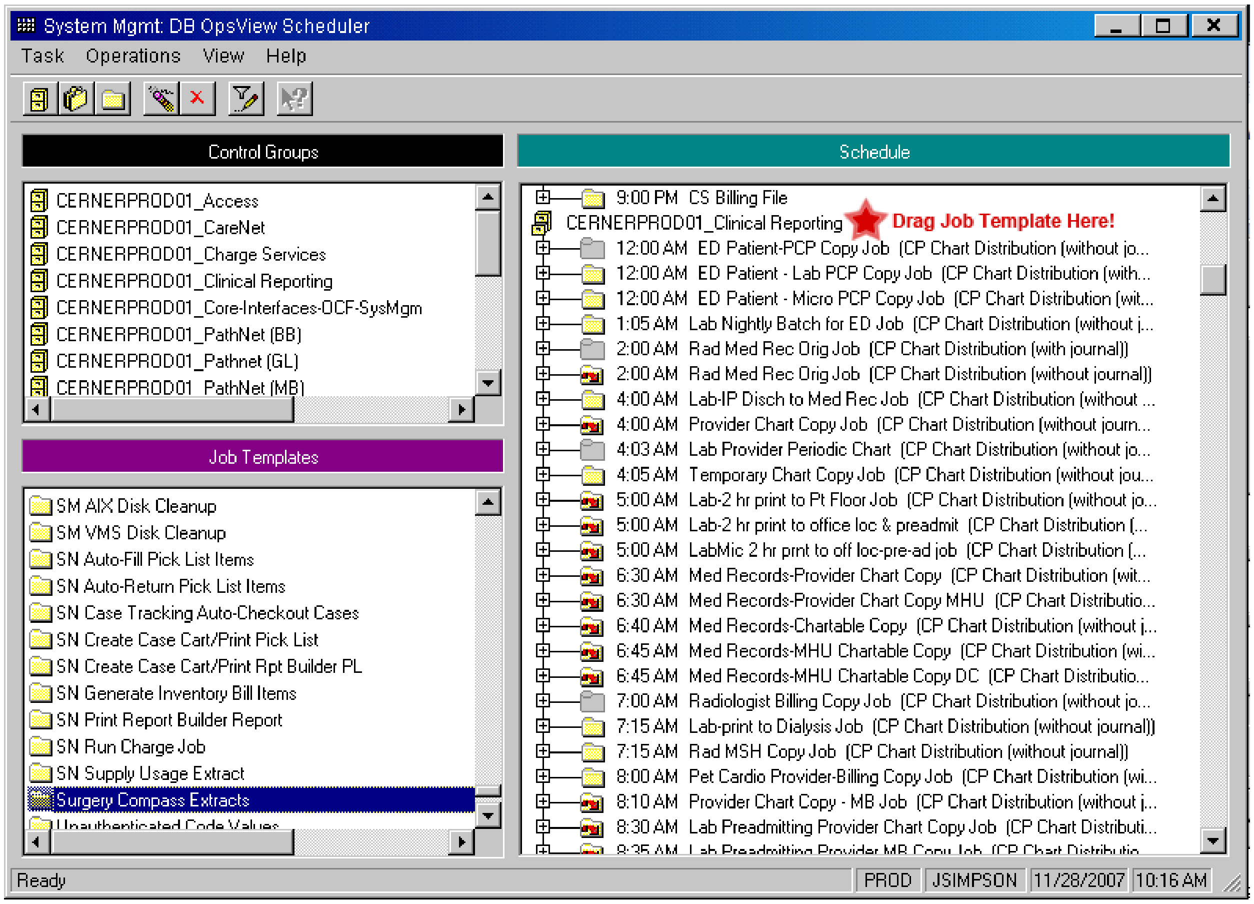
Output distribution:

Operator message:

☐ Select time zone context:

You will be returned to the Job Template – Add screen where you gave your template a name. Shown below the name will be your newly added Step. If you are happy with the results, click the Ok button.

Your job template will now be available in the Job Templates portion of the main screen.



CERNERPROD01_Access
CERNERPROD01_CareNet
CERNERPROD01_Charge Services
CERNERPROD01_Clinical Reporting
CERNERPROD01_Core-Interfaces-OCF-SysMgm
CERNERPROD01_PathNet (BB)
CERNERPROD01_Pathnet (GL)
CERNERPROD01_PathNet (MB)

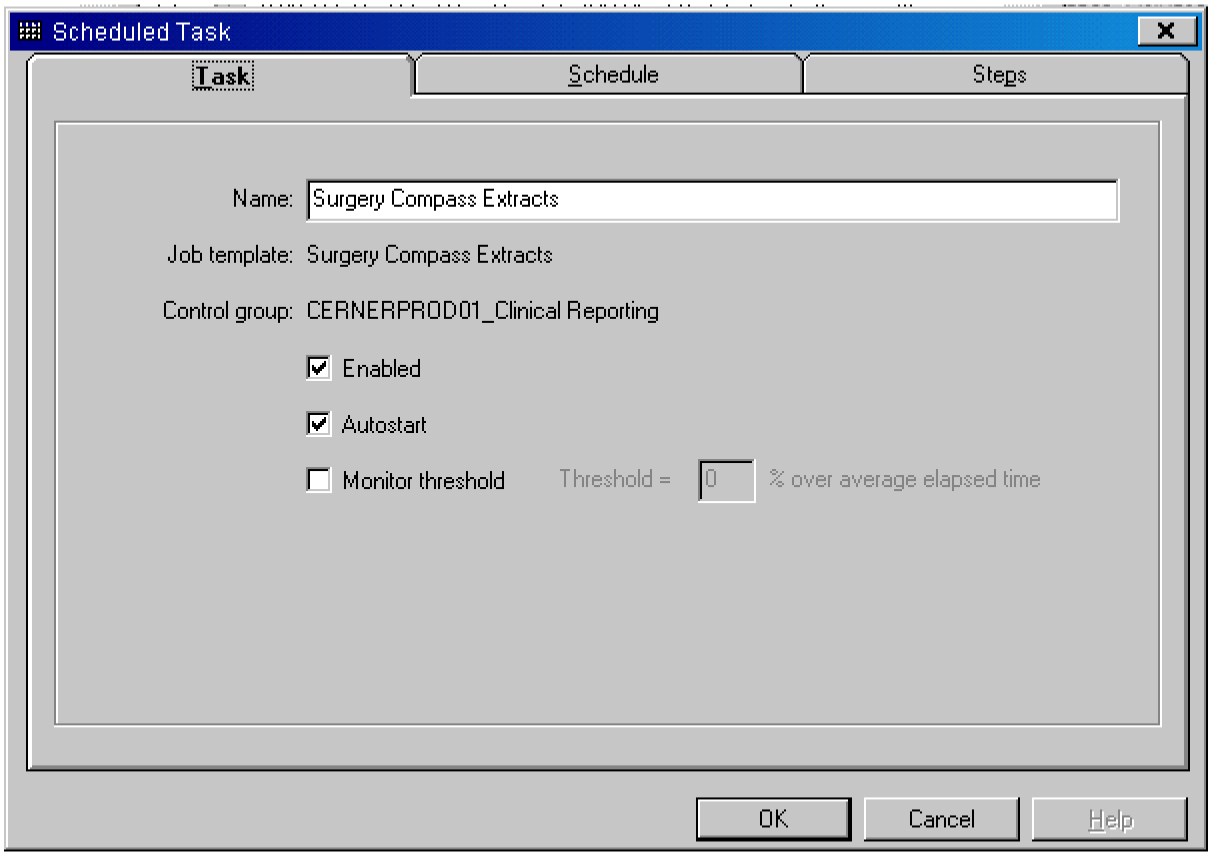
SM AIX Disk Cleanup
SM VMS Disk Cleanup
SN Auto-Fill Pick List Items
SN Auto-Return Pick List Items
SN Case Tracking Auto-Checkout Cases
SN Create Case Cart/Print Pick List
SN Create Case Cart/Print Rpt Builder PL
SN Generate Inventory Bill Items
SN Print Report Builder Report
SN Run Charge Job
SN Supply Usage Extract
Surgery Compass Extracts
Unauthenticated Code Values

9:00 PM CS Billing File
CERNERPROD01_Clinical Reporting
12:00 AM ED Patient-PCP Copy Job (CP Chart Distribution (without jo...
12:00 AM ED Patient - Lab PCP Copy Job (CP Chart Distribution (with...
12:00 AM ED Patient - Micro PCP Copy Job (CP Chart Distribution (wit...
1:05 AM Lab Nightly Batch for ED Job (CP Chart Distribution (without j...
2:00 AM Rad Med Rec Orig Job (CP Chart Distribution (with journal))
2:00 AM Rad Med Rec Orig Job (CP Chart Distribution (without journal))
4:00 AM Lab-IP Disch to Med Rec Job (CP Chart Distribution (without ...
4:00 AM Provider Chart Copy Job (CP Chart Distribution (without journ...
4:03 AM Lab Provider Periodic Chart (CP Chart Distribution (without jo...
4:05 AM Temporary Chart Copy Job (CP Chart Distribution (without jou...
5:00 AM Lab-2 hr print to Pt Floor Job (CP Chart Distribution (without jo...
5:00 AM Lab-2 hr print to office loc & preadmit (CP Chart Distribution (...
5:00 AM LabMic 2 hr prnt to off loc-pre-ad job (CP Chart Distribution (...
6:30 AM Med Records-Provider Chart Copy (CP Chart Distribution (wit...
6:30 AM Med Records-Provider Chart Copy MHU (CP Chart Distributio...
6:40 AM Med Records-Chartable Copy (CP Chart Distribution (without j...
6:45 AM Med Records-MHU Chartable Copy (CP Chart Distribution (wi...
6:45 AM Med Records-MHU Chartable Copy DC (CP Chart Distributio...
7:00 AM Radiologist Billing Copy Job (CP Chart Distribution (without jo...
7:15 AM Lab-print to Dialysis Job (CP Chart Distribution (without journal))
7:15 AM Rad MSH Copy Job (CP Chart Distribution (without journal))
8:00 AM Pet Cardio Provider-Billing Copy Job (CP Chart Distribution (wi...
8:10 AM Provider Chart Copy - MB Job (CP Chart Distribution (without j...
8:30 AM Lab Preadmitting Provider Chart Copy Job (CP Chart Distributi...
8:35 AM Lab Preadmitting Provider MB Copy Job (CP Chart Distributio...

Ready PROD JSIMPSON 11/28/2007 10:16 AM

Next, you need to add your Job Template to the Schedule. Do this by dragging the template to the Control Group title in the Schedule window on the right hand side of the screen. For my example, I am going to drag the Surgery Compass Extracts template in to the CERNERPROD01_Clinical Reporting control group.

Next, you will be presented with the Scheduled Task window. By default, this window shows you the Task Tab which gives you a brief summary of your job.



Name: Surgery Compass Extracts

Job template: Surgery Compass Extracts

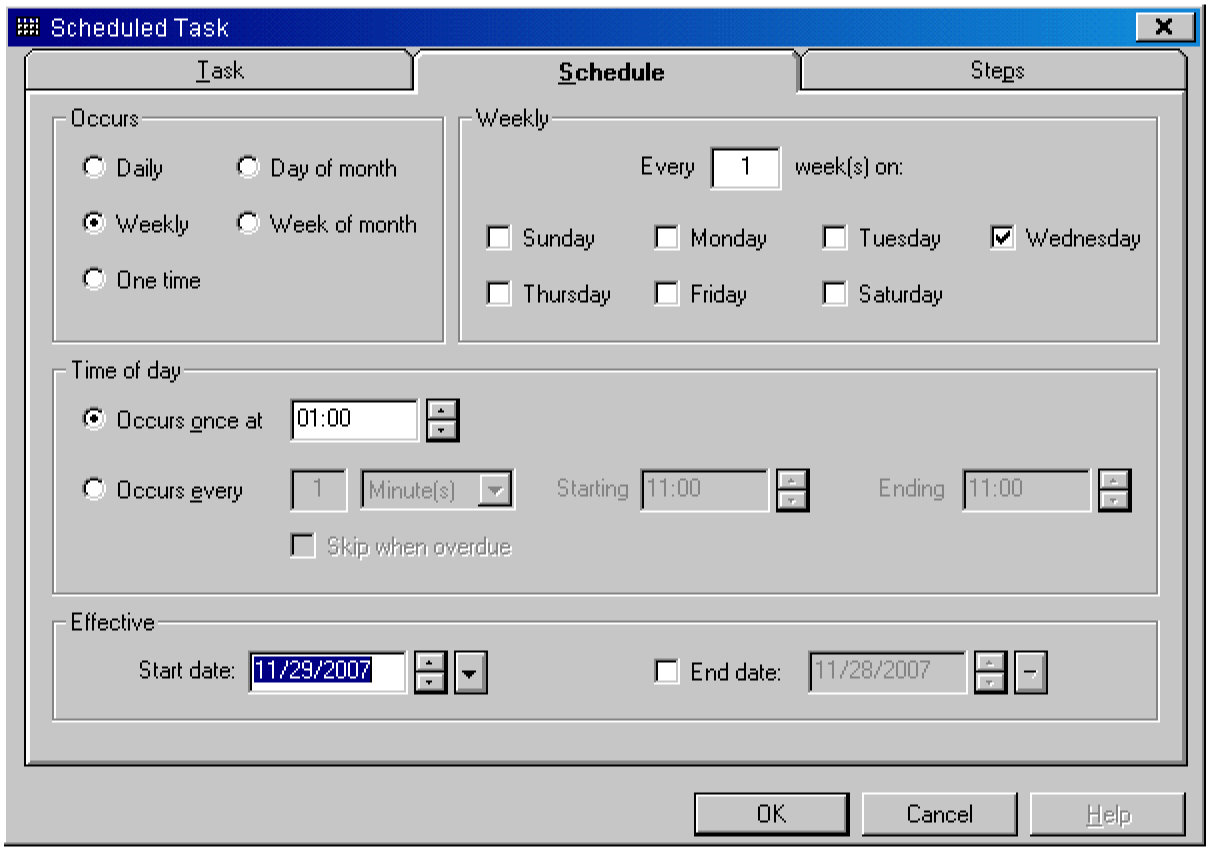
Control group: CERNERPROD01_Clinical Reporting

☑ Enabled

☑ Autostart

☐ Monitor threshold Threshold = 0 % over average elapsed time

Click on the Schedule table to set up the job times and frequency.


Scheduled Task

Task | Schedule | Steps

Occurs: Daily, Day of month, [✓ Weekly], Week of month, One time

Weekly: Every 1 week(s) on: Sunday, Monday, Tuesday, [✓ Wednesday], Thursday, Friday, Saturday

Time of day: [✓ Occurs once at] 01:00

Occurs every 1 Minute(s) Starting 11:00 Ending 11:00

Skip when overdue

Effective: Start date: 11/29/2007 End date: 11/28/2007

OK Cancel Help

In the sample above, I have set my job to run every Wednesday at 01:00. For this project, I wanted to allow a few extra days between the final collection date (last Sunday) and the operations run date for staff to complete any necessary data entry.

A3.1 – Common Tables and Relationships

A3.1.1 Person Management

Figure 1 – Person Data Model

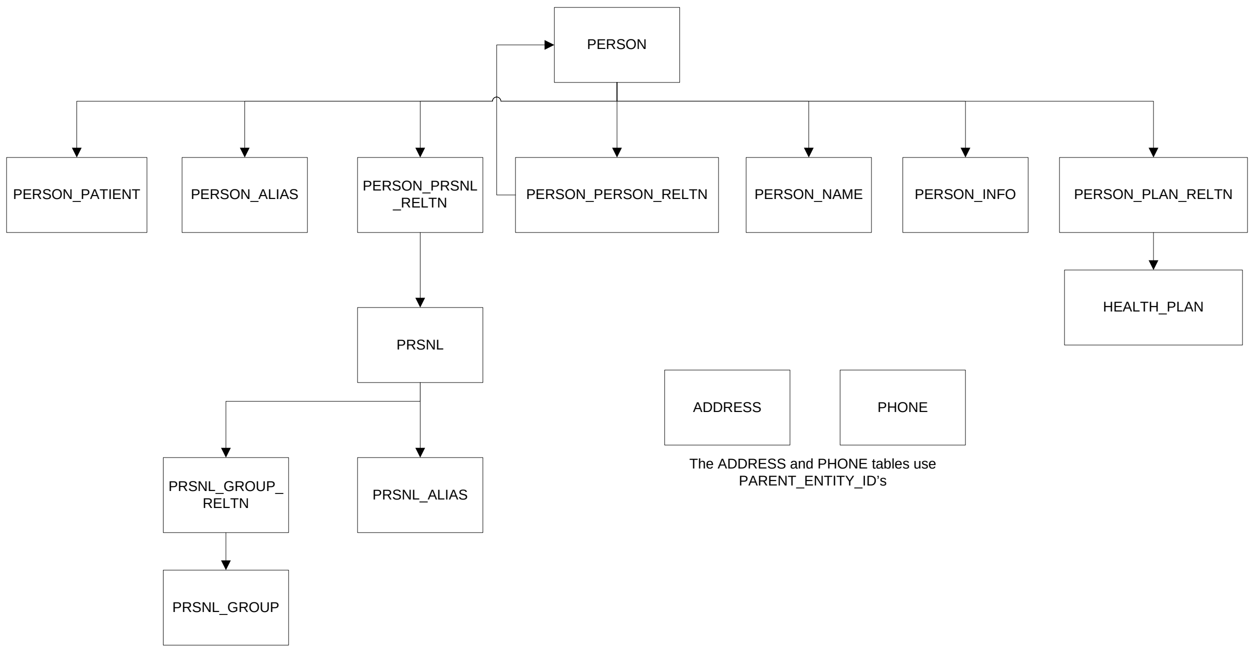
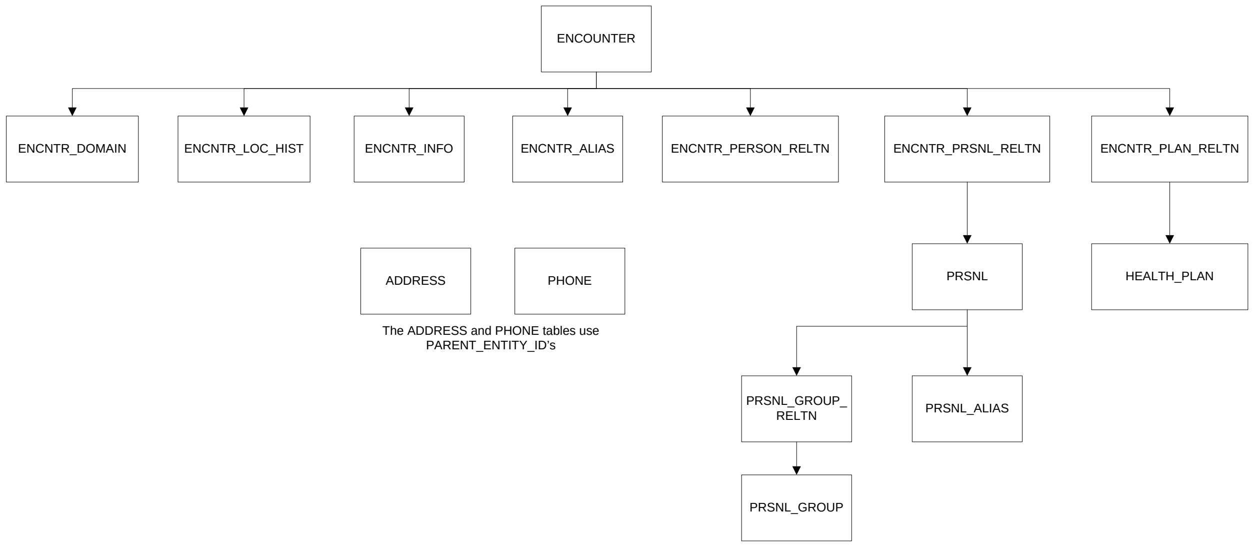


Figure 2 – Encounter Data Model



Person Management Table Descriptions

Table Name	Description
PERSON	Contains information about a person such as current name, DOB, Gender and other non-patient specific details.
PERSON_PATIENT	Contains additional person information for person records which represent patients. Includes fields such as ADOPTED_CD and other patient data.
PERSON_ALIAS	Contains life time aliases for a person. Types of aliases included would be SIN, Provincial Health Card #, SSN and the most current MRN.
PERSON_PRSNL_RELTN	Associates personnel to a patient. These would be lifetime associations such as family physician.
PRSNL	Contains personnel information such as name, position and whether the person is a physician.
PRSNL_GROUP_RELTN	Associates PRSNL records to a common group.
PRSNL_GROUP	Hospital defined groups that can be used to associate physicians together. E.g. Cardiologists.
PRSNL_ALIAS	Aliases associated with a prsnl record. Physician billing number and employee number are examples.
PERSON_PERSON_RELTN	Establishes a lifetime relationship to another person record. Examples would be parents and siblings.
PERSON_NAME	Contains a history of the name of a patient. *Note* If history is turned on, PERSON_NAME_HIST will contain the history instead of PERSON_NAME.
PERSON_INFO	Contains user defined person level fields from the registration conversation.
PERSON_PLAN_RELTN	Establishes a relationship between a person and a health plan.
HEALTH_PLAN	Specific list health plans.
ADDRESS	Addresses (unique addresses are determined by the ADDRESS_TYPE_CD field)
PHONE	Phone Numbers (Unique phone numbers are determined by the PHONE_TYPE_CD field)
ENCOUNTER	Contains patient visit information.
ENCNTR_DOMAIN	Census table which is purged on a regular basis. Can be used to find current census by checking the END_EFFECTIVE_DT_TM field. It is recommended that a custom table be created if historical census reporting is required.
ENCNTR_LOC_HIST	History of locations and medical services that a patient has had during an encounter.
ENCNTR_INFO	User defined encounter related fields from the Person Management conversations.
ENCNTR_ALIAS	Contains encounter specific aliases. The most common values used are MRN and FIN #.
ENCNTR_PERSON_RELTN	Links person records to an encounter. E.g. Patient Accompanied by.
ENCNTR_PRSNL_RELTN	Relates PRSNL records to an encounter. Examples would be

	attending physician.
ENCNTR_PLAN_RELTN	Encounter specific health plans.

Code Sample

Census

Census is primarily driven by the ENCNTR_DOMAIN table. In addition to ENCNTR_DOMAIN, the ENCOUNTER and ENCNTR_LOC_HIST tables must be used. The ENCNTR_LOC_HIST table contains the location and medical service of a patient at a specific time.

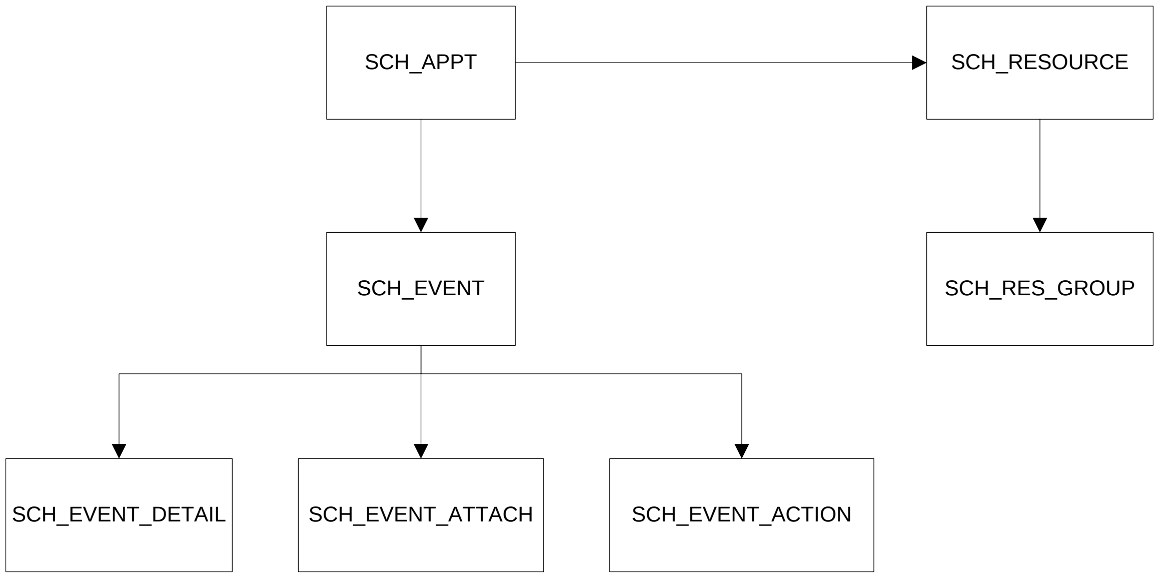
*** NOTE ** The ENCNTR_DOMAIN table is purged on a nightly basis. Every client has a different purge criteria. The average I have seen is that records are purged from ENCNTR_DOMAIN ten days after discharge. This makes reporting a historical census more than ten days back impossible. However, it is very easy to implement a custom table that is populated every minute or so which can handle census reporting as far back as you want to go.*

In the example below, we are looking for the census at the previous midnight. You would normally do a check for patient type but I excluded it to keep this example as clear as possible.

```
SELECT
            NURSE_UNIT  = UAR_GET_CODE_DISPLAY(ELH.LOC_NURSE_UNIT_CD)
FROM        ENCNTR_DOMAIN          ED,
            ENCOUNTER              E,
            ENCNTR_LOC_HIST        ELH
PLAN ED
      WHERE ED.END_EFFECTIVE_DT_TM > CNVTDATETIME(CURDATE, 0)
JOIN E
      WHERE E.ENCNTR_ID = ED.ENCNTR_ID
      AND   E.REG_DT_TM < CNVTDATETIME(CURDATE, 0)
      AND   (E.DISCH_DT_TM > CNVTDATETIME(CURDATE, 0)
            OR
            E.DISCH_DT_TM = NULL)
      AND   E.ACTIVE_IND = 1
JOIN ELH
      WHERE ELH.ENCNTR_LOC_HIST_ID =
                  (SELECT MAX(ELH2.ENCNTR_LOC_HIST_ID)
                   FROM ENCNTR_LOC_HIST         ELH2
                   WHERE ELH2.ENCNTR_ID = ED.ENCNTR_ID
                   AND   ELH2.TRANSACTION_DT_TM < CNVTDATETIME(CURDATE,0))
```

A.3.1.2 – Scheduling

Figure 1 – Appointment Data Model

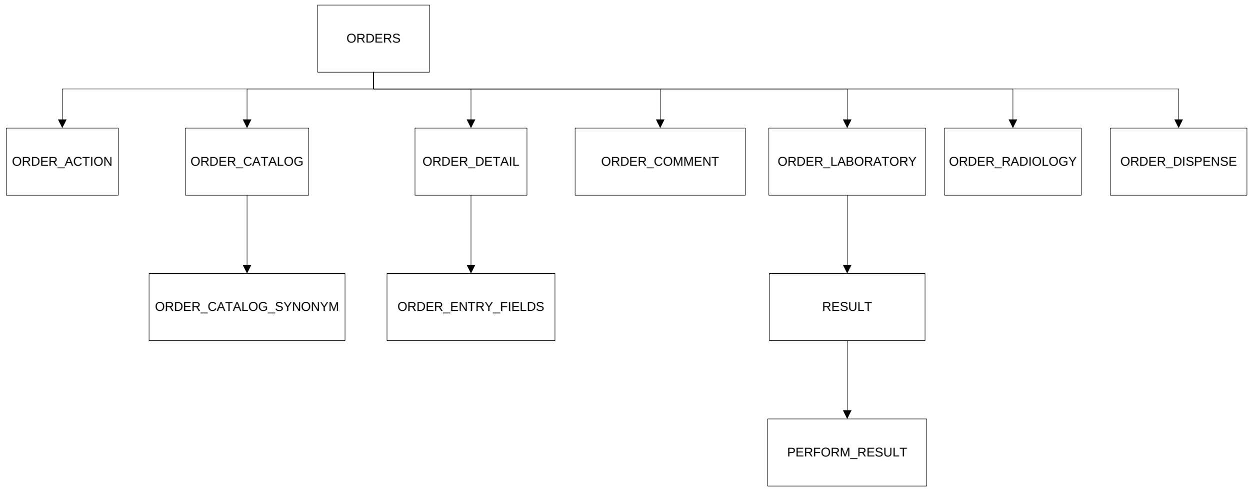


Scheduling Table Descriptions

Table Name	**Description**
SCH_APPT	Contains multiple rows for each appointment and the resources associated with the appointment. There will be a row for the patient and a row for each resource (e.g. physician, surgery location, etc). This table also contains the appointment date and time. All SCH_APPT rows are linked together by SCH_EVENT_ID.
SCH_EVENT	Contains appointment specific information such as appointment type.
SCH_EVENT_DETAIL	Contains user defined appointment fields.
SCH_EVENT_ATTACH	Links orders attached to an appointment.
SCH_EVENT_ACTION	History of actions relating to appointment. Includes bookings, cancels, no-shows, re-schedules.
SCH_RESOURCE	List of resources that can be associated to an appointment.
SCH_RES_GROUP	User defined groups created to group common resources together.

A.3.1.3 – Orders

Figure 1 – Orders Data Model

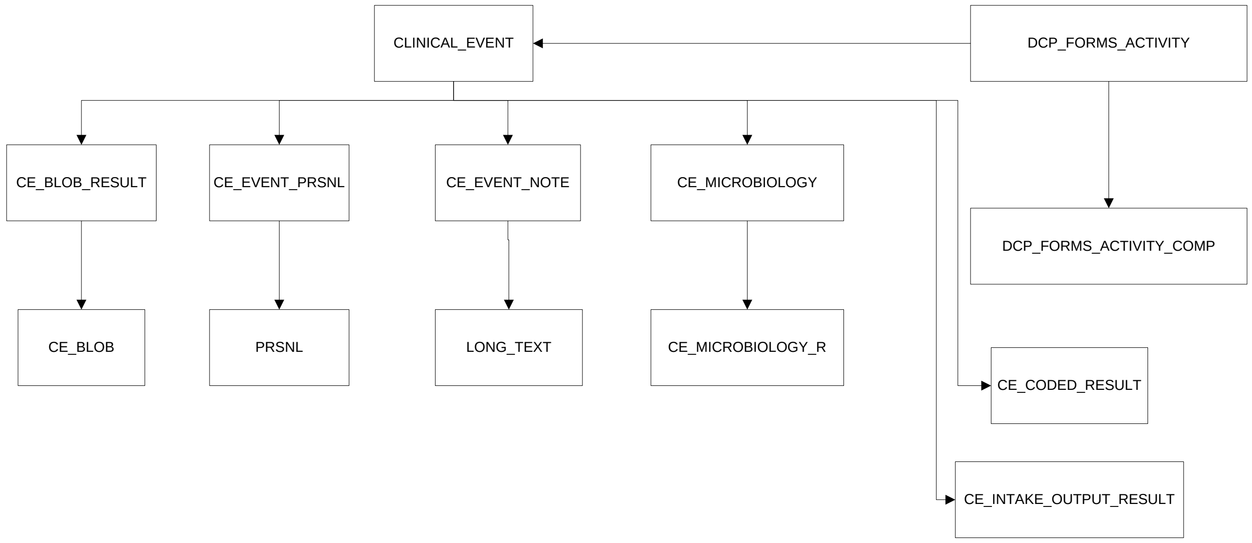


Orders Table Descriptions

Table Name	**Description**
ORDERS	Contains the primary record for an order.
ORDER_ACTION	Details the history of events that an order goes through.
ORDER_CATALOG	Reference list of orderables.
ORDER_CATALOG_SYNONYM	Reference list of alternative synonyms for orderables.
ORDER_DETAIL	Contains the individual details of an order (e.g. Frequency, Stop Date)
ORDER_ENTRY_FIELDS	Reference list of order entry fields used in the order detail table.
ORDER_COMMENT	Contains order comments. Comments are stored in the LONG_TEXT table and the ORDER_COMMENT links the ORDER_ID to the LONG_TEXT table.
ORDER_LABORATORY	Contains additional information for Lab and Microbiology orders
RESULT	Contains the result of a lab order
PERFORM_RESULT	Contains the history of a lab result
ORDER_RADIOLOGY	Contains radiology order information
ORDER_DISPENSE	Contains pharmacy dispenses related to an order.

A.3.1.4 – Clinical Events

Figure 1 – Clinical Event Data Model



Clinical Events Table Descriptions

Table Name	Description
CLINICAL_EVENT	Contains the primary record for a clinical event.
CE_BLOB	Contains BLOB text data
CE_EVENT_PRSNL	Personnel relationships to a clinical event record.
CE_EVENT_NOTE	Notes
CE_MICROBIOLOGY	Contains the Mico Organism
CE_MICROBIOLOGY_R	Assists in linking a ce_microbiology row to a clinical event
CE_CODED_RESULT	Codified results (e.g. check boxes on a powerform)
CE_INTAKE_OUTPUT_RESULT	I&O results reference
DCP_FORMS_ACTIVITY	Top level PowerForm Activity
DCP_FORMS_ACTIVITY_COMP	PowerForm linking to the clinical events

A.3.1.5 – PowerForms

The results of PowerForms are primarily stored in the CLINICAL_EVENT table. However, due to their unique structure multiple joins to the CLINICAL_EVENT are necessary to retrieve all of the PowerForm Data.

At the top level, PowerForm activity is recorded in the DCP_FORMS_ACTIVITY table. This is where the FORM_DT_TM and PowerForm name is stored. Additionally, we are required to link the DCP_FORMS_ACTIVITY_COMP table in order to get our event_id needed to join to the CLINICAL_EVENT table.

The following example will pull an the contents of an entire PowerForm (minus the comments which can be located at any level in the CLINICAL_EVENT table)

```
SELECT INTO "NL:"
       FORM_NAME      = DFA.DESCRIPTION,
       FORM_DATE      = FORMAT(DFA.FORM_DT_TM,"MM/DD/YY;;D"),
       FORM_TIME      = FORMAT(DFA.FORM_DT_TM,"HH:MM;;Q"),
       SECTION        = CE2.EVENT_TITLE_TEXT,
       EVENT_L1       = CE3.EVENT_TITLE_TEXT,
       RESULT_L1      = CE3.EVENT_TAG,
       PARENT_L2      = CE3.EVENT_ID,
       EVENT_L2       = CE4.EVENT_TITLE_TEXT,
       RESULT_L2      = CE4.EVENT_TAG,
       PARENT_L3      = CE4.EVENT_ID,
       EVENT_L3       = CE5.EVENT_TITLE_TEXT,
       RESULT_L3      = CE5.RESULT_VAL
FROM          DCP_FORMS_ACTIVITY                    DFA,
              DCP_FORMS_ACTIVITY_COMP               DFAC,
              CLINICAL_EVENT                        CE1,
              CLINICAL_EVENT                        CE2,
              CLINICAL_EVENT                        CE3,
              CLINICAL_EVENT                        CE4,
              CLINICAL_EVENT                        CE5,
              DUMMYT                                D1,
              DUMMYT                                D2
PLAN DFA
       WHERE DFA.PERSON_ID = nPERSON_ID
       AND   DFA.FORM_DT_TM BETWEEN CNVTDATETIME(cFROM_DATE) AND CNVTDATETIME(cTO_DATE)
       AND   DFA.DESCRIPTION IN ("Patient History","Vital Signs","Adult Ongoing Assessment")
       AND   DFA.ACTIVE_IND = 1
       AND   DFA.FORM_STATUS_CD IN (cvAUTH, cvMODIFIED)
JOIN DFAC
       WHERE DFAC.DCP_FORMS_ACTIVITY_ID = DFA.DCP_FORMS_ACTIVITY_ID
       AND   DFAC.PARENT_ENTITY_NAME = "CLINICAL_EVENT"
JOIN CE1
       WHERE CE1.EVENT_ID = DFAC.PARENT_ENTITY_ID
       AND   CE1.VALID_UNTIL_DT_TM > CNVTDATETIME(CURDATE, CURTIME3)
       AND   CE1.RESULT_STATUS_CD IN (cvAUTH, cvMODIFIED)
JOIN CE2
       WHERE CE2.PARENT_EVENT_ID = CE1.EVENT_ID
       AND   CE2.VALID_UNTIL_DT_TM > CNVTDATETIME(CURDATE, CURTIME3)
       AND   CE2.RESULT_STATUS_CD IN (cvAUTH, cvMODIFIED)
JOIN CE3
       WHERE CE3.PARENT_EVENT_ID = CE2.EVENT_ID
       AND   CE3.VALID_UNTIL_DT_TM > CNVTDATETIME(CURDATE, CURTIME3)
       AND   CE3.RESULT_STATUS_CD IN (cvAUTH, cvMODIFIED)
```

```
JOIN D1
JOIN CE4
       WHERE CE4.PARENT_EVENT_ID = CE3.EVENT_ID
       AND   CE4.VALID_UNTIL_DT_TM > CNVTDATETIME(CURDATE, CURTIME3)
       AND   CE4.RESULT_STATUS_CD IN (cvAUTH, cvMODIFIED)
JOIN D2
JOIN CE5
       WHERE CE5.PARENT_EVENT_ID = CE4.EVENT_ID
       AND   CE5.VALID_UNTIL_DT_TM > CNVTDATETIME(CURDATE, CURTIME3)
       AND   CE5.RESULT_STATUS_CD IN (cvAUTH, cvMODIFIED)
WITH OUTERJOIN=D1, OUTERJOIN=D2, NULLREPORT
```