6

In This Chapter

The Basics

Note

In GoldMine Premium, there is no capability of having a single Lookup.ini per Contact set as you had available with the GoldMine Standard Edition.

Rules

A. The Lookup.ini may not exceed 64K in size (theoretically).

B. No single line in the Lookup.ini may exceed 250 characters.

Personally, I have broken the 64K barrier for clients in the XP Pro environment, and the clients claim that the Lookup.ini does function as expected. Your mileage may vary.

This is going to be an easy chapter to rewrite as nothing has changed between versions and builds of GoldMine Premium with regard to this area. The Lookup.ini is how FrontRange Solutions allows the GoldMine Administrators to add external functionality to their GoldMine product. The Lookup.ini is an extremely powerful programming tool that is, unfortunately, not used to its fullest advantage, if at all, in most GoldMine implementations. I can not stress enough, how much control you can gain over your GoldMine with the proper programming of this utility, and, if you have not already guessed, this is my favorite chapter in the entire book. I just love writing this chapter. I hope that you enjoy reading it just as much.

One could write a program that would automatically assign a unique account number to each and every record as it is being created. GoldMine does not have a radio button option, however, you could program (emulate) one into GoldMine via the Lookup.ini. How about the capability to automatically assign new leads to account managers in a rotational manner? This is easily accomplished through the use of the Lookup.ini functionality within GoldMine Premium. Would you like to run an external application, or play a macro when a field changes, when a new contact history record is added, or when a calendar record is edited? You can do all of this as well with the proper programming of your Lookup.ini. How would you like a consistent color coding schema throughout the corporation for calendar activities? Yes, this too is even possible through the Lookup.ini. If you can imagine it, and if it is not already in the GoldMine application, you could probably program it in through the Lookup.ini.

Here is one that I am including in this version of The Hacker's Guide. How would you like to give your users the option of entering **Mr. & Mrs. DJ and Carol Hunt Jr.** into the Contact field, and having the Dear field populated with Dr. & Mrs., the Contact field populated with DJ Hunt Jr., the LastName field populated with Hunt, and finally the Spousal field populated with Carol Hunt?

When thinking of the Lookup.ini, think **Power**.

First of all, it should be understood that there is no Lookup.ini distributed with the GoldMine product. The Lookup.ini is an application that one must create from the get go. I recommend that you create the Lookup.ini using the Windows NotePad application (**Start** | **Run** | **NotePad**). Once created in NotePad, it must be saved as Lookup.ini (see sidebar Tip on the next page) in the root folder of your GoldMine.

The Lookup.ini is broken down into sections. Some of the sections are self contained while other sections are dependant upon the instructions in the **[AutoUpdate]** section of the Lookup.ini. Each section is defined by the header which is defined with enclosed square brackets. Let's start with the **[AutoUpdate]** section which would have a header:

```
[AutoUpdate]
```

I should explain that throughout this chapter I will use white space, and comments freely to assist in the readability of the Lookup.ini application. If you have an exceptionally large Lookup.ini, you will have to consider the rules that must be adhered to (see sidebar Rules) regarding size limitations, and you may have to forego readability formatting for size considerations.

So, what is contained in the **[AutoUpdate]** section? The **[AutoUpdate]** section defines the fields that GoldMine is to watch for changes, and contains a pointer to the section(s) that are to be processed should one of those fields being watched change. There is a single exception to the watch side of the equation, and that is GoldMine allows the **[AutoUpdate]** section to also watch for the creation of a new contact record as opposed to a field, and when a record is created the right side of the equation defines the section(s) that are to be processed accordingly.

Rules

C. The Lookup.ini must reside in the same directory as the GoldMine Premium executable GMW.exe.

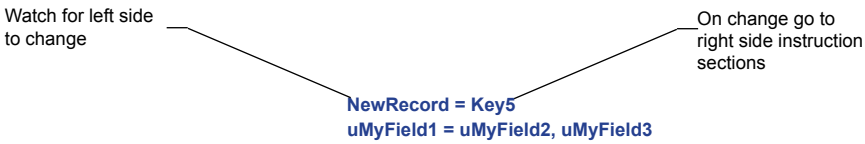
D. You may only have one [AutoUpdate] section in your Lookup.ini.

Tip

*If you are using NotePad to create/edit your Lookup.ini, make certain, when you save the file, that the .txt extension is not put on the file automatically by NotePad. The file name **must** be **Lookup.ini**. If your resulting file is named **Lookup.ini.txt**, you must rename the file back to Lookup.ini.*

Note

The Lookup.ini functions equally well for the Microsoft SQL or the Firebird SQL versions of GoldMine, although I must warn you that I have heard rumors of FrontRange dropping support for Firebird.



If you are planning on formatting a field in a certain way, you can ask GoldMine to watch the field, and then go to the same named section to process the instruction set contained there in. An example of that would be:

```
Key4 = Key4
```

You may wish to add comments to your Lookup.ini. Programmers do this a lot for readability, and for future understanding of what the section was expected to do. If you wish to add a comment to your Lookup.ini, you precede the line with a semicolon (**;**).

Here is an example of what this much of a Lookup.ini might look like:

```
[AutoUpdate]

  ;Watch for a new record being created and process the instruction
  ;set in the Key5 section.

  NewRecord = Key5

  ;Watch for Key4 being changed, and process the instruction sets for
  ;Key4, Key5, and uMyField1 in that order.

  Key4 = Key4, Key5, uMyField1
```

Under the instruction set sections, you must define the action which is to be performed. If the instruction set pertains to a field, then the header must be the name of the field. In our example above, there must be an instruction set for the **Key4**, **Key5**,and **uMyField1** fields or nothing will be processed for the missing instruction sets.

The instruction set itself has sections. In the first section are the lookups. Each lookup is numbered sequentially. In example: **Lookup1**, **Lookup2**, **Lookup3**, ... When processing begins, each lookup will be considered in its sequential order, and all lookups will be considered until one of those lookup variables produces a positive result, or all have been processed with negative results. Consider each of these lookups to be the names of variables. The variable name is on the left side of the equation while the function is on the right hand side of the equation.

Under each lookup is a value list to compare against. If the value in the Lookup(x) variable matches one of the values in the comparison list, the function to the right of the equal sign in the equation is processed. If no match is found, the next lookup variable is evaluated and so on in turn.

Should all of the lookups fail to produce a match, then there can be a fail safe statement. This is called the **Otherwise** statement. If Lookup1 produces a match, process the appropriate expression. If Lookup2 produces a match, then process the appropriate expression. Otherwise, if all Lookup(x)s have returned False, process this expression. You may use the Otherwise statement as a self standing statement without having any Lookup statements.

Note

*Numeric based fields always contain a value. If you are updating a numeric based field through the Lookup.ini, then you **must** include **Overwrite = 1** in the action section.*

The last section is the **Overwrite** statement. This statement is a switch which instructs GoldMine, if there is already information contained in this field, to overwrite the field information with this new information or not depending on the setting of the switch. A zero (**0**) tells GoldMine that it **is not okay** to overwrite the information contained in the field, while a one (**1**) tells GoldMine that it **is permitted** to overwrite the field with the new information.

Updating Fields

In this section, I will put the previously disclosed information into practical use. I will demonstrate a Lookup.ini to update the **Key5** field with a unique account number each time a new record is created. I will populate the **Key1** field with the Account Managers name, and I will populate a user defined field, **uUserName**, with the GoldMine login name for that Account Manager.

Prior to beginning with the Lookup.ini, we need to make some corporate decisions. The **Key5** field must have its **Read Access** attribute set to **(public)** while its **Update Access** attribute is set to **MASTER** or the GoldMine Administrators UserID. I would further stipulate that the user who is creating the new record will be the Account Manager. Lastly, there must be a user defined field created that is called **uUserID**, be a character based field, and be 8 characters long.

When creating the unique account number for each record, one must consider the possibility that synchronization may be occurring. If the field were populated with just a number, then there is the chance for duplication if records are created Server-side and Remote-side as the Counter variable is stored in the Lookup table which synchronizes between the two locations. As I am looking for uniqueness, I must append something to the number, such as the UserID, which itself is unique, that will cause the account number to maintain its uniqueness (most of the time at least).

```
[AutoUpdate]
  NewRecord = Key1, Key5, uUserID

[Key1]
  Lookup1 = &UserName

  DJ = DJ Hunt
  BOB = Bob Jefferson
  BUBBA = Bubba Gump
  Otherwise = Up for grabs

  Overwrite = 1

[Key5]
  Otherwise = &trim(Contact2->uUserID)+padl(ltrim(str(counter([AcctNo],1))),10,[0])

  Overwrite = 1

[uUserID]
  Otherwise = &UserName

  Overwrite = 1
```

Let's take a look at the Lookup.ini section by section. The first section is:

```
[AutoUpdate]
  NewRecord = Key1, uUserID, Key5
```

In this section, the **[AutoUpdate]** section, I am saying that whenever there is a **NewRecord** created that GoldMine should process the instruction sets under the sections **Key1**, **Key5**, and **uUserID**. Just as important, the instruction sets **must** be processed in that specified order. This is done in case the processing in one instruction set is dependant upon the results of the processing in a previous instruction set, better known as cascading instruction sets.

The first instruction set to be processed after the creation of a new record is the **Key1** instruction set.

```
[Key1]
  Lookup1 = &UserName

  DJ = DJ Hunt
  BOB = Bob Jefferson
  BUBBA = Bubba Gump
  Otherwise = Up for grabs

  Overwrite = 1
```

I am using one lookup variable in this instruction set, and I am setting that **Lookup1** variable equal to the resulting value of the GoldMine macro **&UserName**. Here I have used one of the many GoldMine developed macros. Please refer to Appendix B at the back of this book for other available and valuable GoldMine macros. This particular macro will return the logged in users GoldMine **UserID**.

In the next three lines of this instruction set, I am comparing the left side of the equation to the information stored in the **Lookup1** variable. Therefore, if the information in **Lookup1** matches either **DJ**, **BOB** or **BUBBA**, then the value on the right hand side of the appropriate equation should be pushed into the **Key1** field.

Should neither **DJ**, **BOB** or **BUBBA** match the value in the **Lookup1** variable, I have included an **Otherwise** statement. My **Otherwise** statement evaluates if there are no matches found against the value contained in the **Lookup1** variable, and will push **Up for grabs** into the **Key1** field.

Do I require an **Overwrite = 1** statement? Where as, in the past I would have said No, in GoldMine Premium I must answer this question with a definitive Yes. In the past, all but a few of the fields would have been .null. (empty) when a New Record was created within GoldMine. In GoldMine Premium, one now has the ability to input information into any field that exists for the record, hence, for GoldMine Premium and into the GoldMine future, you must utilize the Overwrite statement to account for the possibility that your user may have entered a value into the field.

Emulating Radio Buttons

As you know from a previous chapter, GoldMine only allows the user to add user defined fields in three data types, **Character**, **Numeric** or **Date**. Here's a bit of code that will let you emulate radio buttons within your GoldMine Premium environment using character based fields.

The follow prerequisites are required. You are required to add three new GoldMine fields each being character type, and 1 character in size. They must be named **uRB1**, **uRB2**, and **uRB3** respectively. You should place these on a screen and adjust the **Field Label Size** to **0** while the **Data Size** need only be **2** characters wide. For clarity, you should add an expression field to the right of each field so that it will be easily understood as to which radio button has been selected. You should set up the lookups for these fields to not Allow blank input, and to Pop-up when selected. Additionally, I include one lookup entry, **~chr(169)**, in each F2 Lookup list. A sample of our GoldMine representation is shown here in Figure 6-1.

```
[AutoUpdate]
  uRB1 = uRB2, uRB3
  uRB2 = uRB1, uRB3
  uRB3 = uRB1, uRB2

[uRB1]
  Otherwise = &space(0)

  Overwrite = 1

[uRB2]
  Otherwise = &space(0)

  Overwrite = 1

[uRB3]
  Otherwise = &space(0)

  Overwrite = 1
```

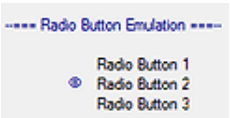


Figure 6-1

Having set those prerequisites, let's look at the individual sections. The last three sections are similar with only variations in field names, therefore, I will only examine the first.

```
[AutoUpdate]
  uRB1 = uRB2, uRB3
  uRB2 = uRB1, uRB3
  uRB3 = uRB1, uRB2
```

The **[AutoUpdate]** section is very straight forward. The first line is instructing GoldMine to watch the **uRB1** for any change, and if it changes, to follow the instruction sets under the **uRB2** and **uRB3** sections in that specific order. The other two lines are just variants of the first so I won't described those in detail.

Now let's examine the instruction set under **uRB2**.

```
[uRB2]
  Otherwise = &space(0)

  Overwrite = 1
```

Tip

Should you ever need to blank out a date based field through the use of the Lookup.ini, I have found that this works nicely.

&.null.

I do not require a **Lookup1** variable field. Not having a **Lookup1** statement causes the processing to drop immediately to the **Otherwise** statement which states: replace the **uRB2** field with nothing (not to be confused with a **.null.** value). I am forcing the value of nothing employing the **space()** function, and I am saying that there should be **0** spaces inserted into this field.

Finally, just in case there was previously a value in the **uRB2** field, I must include the **Overwrite** statement. You'll remember that one (**1**) in this statement lets GoldMine know that it is acceptable to overwrite the old value with the new value which, in this case, is **space(0)**.

Rotationally Assigning Leads to Representatives

Wow! Big section title, huh? Here is the scenario for this Lookup.ini function. This particular office has twelve sales representatives. As new leads are entered into GoldMine, the leads should be distributed to each representative in turn. I have heard this type of distribution of leads to be called a "Round Robin". This is very simple to accomplish using the power of the Lookup.ini, but of course everything is simple once you know how to do it.

The prerequisite for this is to have created a user defined field, **uNextRep**, **N**, **2**, **0**. You do not have to display this field on any screen as we will only be using this field to store the next representatives number. With the field created, we can now discuss the operation of this Lookup.ini.

```
[AUTOUPDATE]
      NewRecord =  uNextRep,  Key1

[UNEXTREP]
      Lookup1 = iif(counter([NextRep], 0) = 12, [A], [Z])

      A = &counter([NextRep], 1, 1, 1)
      Z = &counter([NextRep],1)

      Overwrite = 1
```

```
[KEY1]
  Lookup1 = uNextRep

  1 = DJ Hunt
  2 = Davey Crockett
  3 = Daniel Boone
  4 = Jimney Cricket
  5 = Donald Duck
  6 = Daisy Duck
  7 = Mickey Mouse
  8 = Minnie Mouse
  9 = Goofey
  10 = Pluto
  11 = Pinnochio
  12 = Gippetto

  Overwrite = 1
```

As you can see in the **[AutoUpdate]** section, above, I am only processing these instruction sets upon the creation of a new record at this time. The first instruction set that I process is the **uNextRep** instruction set (sequencing order is critical here). In that instruction set, I set one lookup variable using the **i**mmediate **if** function. This functions syntax is:

```
iif(<Expression>, <True>, <False>)
```

When using the **iif()** function, you must have an expression that can be evaluated to a logical True or False. If the expression evaluates to a True, then the value from the True position is stored in our Lookup1 variable. Alternatively, if the expression evaluates to a False, then the value from the False position is stored in our Lookup1 variable. Our expression is:

```
Lookup1 = iif(counter([NextRep],0) = 12, [A], [Z])
```

Note

*It may be of interest to you to know that all **counter()** variables are stored in the **Lookup** table. The variable name is stored in the first 9 positions of **Lookup.FieldName** with a **C** in the 10th position. The actual counter value is stored in the **Lookup.Entry** field.*

At the time of processing for this instruction set, I am looking to see if the value in the **counter([NextRep],0)** is equal to **12**, the maximum number of representatives for our rotation. If it is equal to **12**, then it is **True**, and I am stuffing our **Lookup1** variable with an **A**. If the value in the **counter([NextRep],0)** is not equal to **12** then I stuff our **Lookup1** variable with a **Z**. As the expression for **Lookup1**, **counter([NextRep],0)** adds **0** to the **NextRep** variable to return the existing value contained in the NextRep variable, hence, I don't have to worry about incrementing the NextRep variable number in the **Lookup** table when I am just testing for what the actual value is presently.

Next, and first in our comparison list, is the comparison statement if the **Lookup1** variable contains an **A**.

```
A = &counter([NextRep], 1, 1, 1)
```

If the **Lookup1** variable contains **A**, then I employ the counter function to reset our counter value, **NextRep**, to its starting position of **1**. My second comparison employs the counter function again, however, this time the counter function is set to increment my counter value, **NextRep**, by **1**. In example, if my counter value were **2**, it would then be incriminated to **3**, and that value would be stuffed into the **uNextRep** field.

```
Z = &counter([NextRep],1)
```

Note

*For a more detailed explanation of the **Counter()** function, refer to Appendix A.*

Notice the difference between the two counter functions. The syntax for the counter function is:

```
counter(<String Variable>, <Increment>, <Start>, <Action>)
```

The Start and Action parameters are optional, and as was shown in the **Z** statement previously. When the Action parameter is set to **1**, then the counter is reset to the number in the Start parameter. When the Action is set to **2**, the counter will be deleted from the Lookup table where the variable, and value of the counter are stored.

Once I have set the **uNextRep** field to a value, I have completed this instruction set. Although I haven't mentioned the last statement in this instruction set, you should remember, from previous examples, its meaning.

According to the **[AutoUpdate]** set, the next instruction set to be processed is the **Key1** instruction set. The **Key1** instruction set is very simple. I stuff the value stored in **uNextRep** into our **Lookup1** variable, and then use that as a comparison value to our list. In example, if the number in **uNextRep** were **4**, then **Jimney Cricket** would be stuffed into the **Key1** field.

As defined, this Lookup.ini will now assign a new contact record to each representative, in turn, until the value in **uNextRep** reaches **12**. It will then reset that number to **1**, and begin the entire distribution cycle again, hence the term, Round Robin.

Last Name Conversion

I am constantly amazed at all of the control that the end user really has at their disposal for the GoldMine product, and how little of it is employed. If you can dream it, you can probably accomplish it through the use of the Lookup.ini.

Let's look at another example now that we're on a roll. This is an example that I wrote many years ago, and FrontRange liked it so much they added to their FAQ's system, albeit inaccurately. The main screen in GoldMine contains a **Contact1.Contact** field, and a **Contact1.LastName** field. For years GoldMine has incorporated the functionality to strip the last name from the **Contact1.Contact** field, and place it into the **Contact1.LastName** field. For all of those years, it has functioned properly in only about 95% of the cases. The functionality, as it exists in GoldMine, is to trim all of the trailing spaces off of the **Contact1.Contact** field. GoldMine then takes everything from the first space in the **Contact1.Contact** field, when reading from right-to-left, to the end of the **Contact1.Contact** field. GoldMine would then put the resulting value into the **Contact1.LastName** field. For a contact name like DJ Hunt, the value of Hunt would be inserted into the **Contact1.LastName** field. However, if that person happened to be named DJ Hunt Jr., then Jr. would be placed into the **Contact1.LastName** field. Not exactly what one would want, is it? This problem occurs on Jr, Sr, II, III, Esq, Rev, and many other variations of the adjective.

As the correct functionality is not within the GoldMine product itself, I must bring the Lookup.ini into play to rectify the situation. Below is my example of a Lookup.ini that will rectify this issue.

```
[AUTOUPDATE]
  NewRecord =  LastName
  Contact = LastName

[LASTNAME]
  Lookup1 = iif([,] $ Contact, [A], [Z])
  Lookup2 = iif((upper(trim(LastName)) == [JR] .or. upper(trim(LastName)) == [JR.]), [B], [Z])
  Lookup3 = iif((upper(trim(LastName)) == [SR] .or. upper(trim(LastName)) == [SR.]), [B], [Z])
  Lookup4 = iif((upper(trim(LastName)) == [II] .or. upper(trim(LastName)) == [III]), [B], [Z])
  Lookup5 = iif((upper(trim(LastName)) == [ESQ] .or. upper(trim(LastName)) == [ESQ.]), [B], [Z])

   A = &alltrim(substr(Contact1->Contact, rat([ ], substr(Contact1->Contact, 1, rat([,], trim(Contact1-
       >Contact))-1))+1, rat([,], trim(Contact1->Contact)) - rat([ ], substr(Contact1->Contact, 1, rat([,],
       trim(Contact1->Contact))))-1))
   B = &alltrim(substr(Contact1->Contact, rat([ ], substr(Contact1->Contact, 1, rat([ ], trim(Contact1-
       >Contact))-1))+1, rat([ ], trim(Contact1->Contact)) - rat([ ], substr(Contact1->Contact, 1, rat([ ],
       trim(Contact1->Contact))-1))))
  Otherwise = &LastName

  Overwrite = 1
```

While this may appear quite complicated, if we examine it in sections, you will see that it is really very straight forward, and easily understood. The first thing that I am accomplishing is to tell GoldMine when to apply the instruction set for the **LastName** field, and that is, quite simply, whenever a **New-Record** is created or whenever the **Contact** field is changed.

Now I get into the meat of this Lookup.ini, the instruction set for the **LastName** field. You will notice that I have used **5** Lookup variables, the last 4 of which are compound iif() functions. Let's examine Lookup1 first as it will be the first one evaluated.

```
Lookup1 = iif([,] $ Contact, [A], [Z])
```

Here I am looking for an occurrence of a comma (,) within the **Contact** field, and if one is so contained, then to stuff an **A** into the **Lookup1** variable. Otherwise place a **Z** into the **Lookup1** variable. If **Lookup1** contains **A**, then GoldMine will compare that to our list and process the instruction set associated with that value. I will discuss this instruction set later in this section. On the other hand, if the **Lookup1** variable contains a **Z**, then GoldMine will not be able to find a comparison match, and upon reaching the end of the instruction set, will loop back to the beginning. At this point GoldMine would begin evaluating the **Lookup2** variable. With minor variations, **Lookup2** through **Lookup5** are identical, and, consequently, I will only need to look closely at the **Lookup2** variable. You are expected to extrapolate the information that you will require to complete **Lookup3** through **Lookup5** from this explanation.

```
Lookup2 = iif((upper(trim(LastName)) == [JR] .or. upper(trim(LastName)) == [JR.]), [B], [Z])
```

Remembering the syntax of the **iif(<Expression>, <True>, <False>)** function, the expression that I am evaluating, and that must return a logical True or False, is a compound expression. Here is the expression:

```
(upper(trim(LastName)) == [JR] .or. upper(trim(LastName))==[JR.])
```

You will notice that there are, in fact, two expressions, and if either of them returns a True, then the entire compound expression is considered to be True because it is an .or. condition. Whereas, if both

Note

*Everything in statement **A** and **B** comparison is a continuous line of instruction and is only wrapped here for presentation purposes.*

Note

*It is important that you understand that the **Lookup1** variable is evaluated first, and only if unsuccessful will **Lookup2** be evaluated, and so on in succession until all of the lookup variables have been evaluated. If all have failed, and if an **Otherwise** clause has been incorporated, then, and only then, will the **Otherwise** clause be evaluated.*

expressions return a False, then the entire compound expression is considered to be False. Now let's examine the first of the two expressions.

```
upper(trim(LastName)) == [JR]
```

Working outward from the inner most parenthesis, I begin by trimming the spaces from the GoldMine **LastName** field using the **trim()** function. Knowing that everyone will type in a different variation of the value in the **LastName** field, I then force the remaining characters to upper case employing the **upper()** function to accomplish this. One might ask here, why am I examining the contents of the **LastName** field? To answer that, I need to look at the sequence of events that are fired by GoldMine. Let's say that you change the name in the **Contact** field. GoldMine will first write this information into the **Contact** field of the record. GoldMine will then proceed to evaluate its own formula for stripping out the last name. This, then, writes the GoldMine interpreted value into the **LastName** field of the record. Lastly, your Lookup.ini will now be evaluated, therefore, instead of trying to find the last name in the **Contact** field as GoldMine would, I can simply look to the **LastName** field for this value.

Next is my operator. Many of you will think that this is a typo, however, it is not. I have used the double equal sign (**==**) purposely. In dBase syntax, the double equal sign operator has special significance. The double equal sign means **exactly equal to**. Well, what is the difference between exact equal to, and equal to? That's a very tough explanation. The former compares the left and right side of the operator at the byte level, therefore, everything on both sides must be equal. That includes the length, and any spaces. This may have been overly cautious on my part as I had already trimmed off all of the spaces, yet I would rather err on the side of caution. The latter compares the left and right sides at the character level. **Test** with three trailing spaces would return a **True** when compared to **Test** with five trailing spaces.

Lastly is the right hand side of my expression, and this contains the value that I will be comparing against, **JR** in this case. The other half of the compound expression has the right side of the expression **JR.** comparing against the left side of the expression. I have tried to capture all of the possible variants that an end user might enter junior into the **Contact** field. To give you some possibilities for your understanding of this scenario, one might expect to see junior expressed as **jr**, **jr.**, **Jr**, **Jr.**, **JR**, **JR.**, **jR**, and **jR.** to name but a few variants. All that I am attempting to do in my expression is to level the playing field, so to speak, such that the expression can be evaluated fairly, and with some expectation of finding matches. You must remember that the left side of the equation is evaluating to upper case, hence, the right side of the equation must show comparison values in upper case as well.

Should my expression contain either **JR** or **Jr.** then stuff a **B** into the **Lookup2** variable. Otherwise place a **Z** into the **Lookup2** variable. If **Lookup2**, **Lookup3**, **Lookup4** or **Lookup5** contain **B** then GoldMine will compare that to our list, and process the instruction set associated with the **B** value. I will discuss this instruction set in detail later in this section. On the other hand, if the **Lookup2**, **Lookup3**, or **Lookup4** variable contain a **Z**, then GoldMine will not find a comparison match, and upon reaching the end of the instruction set, will loop back to the beginning until it has processed through our list comparing the final variable, **Lookup5**.

It is about time for us to examine my list of comparison values. Naturally the first is **A**, and that instruction set looks like this:

```
A = &alltrim(substr(Contact1->Contact, rat([ ], substr(Contact1->Contact, 1, rat([,], trim(Contact1->Contact))-1))+1, rat([,], trim(Contact1->Contact)) - rat([ ], substr(Contact1->Contact, 1, rat([,], trim(Contact1->Contact))))-1))
```

Note

Remember that this instruction is on one continuous line, and wrapped here for presentation only. This line is under the 250 character per line limitation.

Note

*I'll be the first to admit that, after all these years, I have devised easier functions to handle this, however, as this is the one that FrontRange has posted on their site, I thought that I should post the corrected **Lookup.ini** here.*

The left hand side of the equation, again, is our comparison value, and, if the **Lookup1** variable should match this value, then the expression on the right hand side of our equation would be processed. To try and go through each section of this expression could get rather confusing so I will use pseudo code to try to explain what the function is accomplishing. Mostly, I am using a combination of the **substr(<Field> , <Start Position>, <Number of Characters>)** function, and the **rat(<Character to Look For>,<String to Look In>)** function (refer to Appendix A). I am using these to locate the numerical position of the comma in the **Contact** field, and then to strip off from that position to the end of the string. Here is an example of what might be contained in the **Contact** field.

```
Donald J. Hunt, Pastor
```

In the example above you can count, and see that the comma occurs at position **15**. Don't forget to count those spaces and that decimal point. Since the comma occurs at position **15**, I would want to strip off beginning from position **15** on up (I don't want that comma and the space at position **15** and **16** respectively) to the end of the character string. In this example, that would be 8 characters. My substring function would then look like this after all of the internally parentheses had been processed:

```
substr(Contact1->Contact, 15, 8)
```

In the **B** comparison I am doing virtually the same thing only it is a bit more complicated because I do not have a unique character to search for in our character string. I must, therefore, find the numerical position of the first occurrence of a space reading from the right hand side of our character string, and moving to the left until that position is located.

```
B = &alltrim(substr(Contact1->Contact, rat([ ], substr(Contact1->Contact, 1, rat([ ], trim(Contact1->Contact))-1))+1, rat([ ], trim(Contact1->Contact)) - rat([ ], substr(Contact1->Contact, 1, rat([ ], trim(Contact1->Contact))-1))))
```

Again I am using mainly a combination of the **substr()** function and the **rat()** functions to achieve this goal. Looking at another example of the **Contact** field contents:

```
Donald J. Hunt Jr.
```

We can see that my final expression would be:

```
substr(Contact1->Contact, 15, 4)
```

That's a far cry from the 234 character expression shown above, but it is in effect what I am accomplishing in the expression. I wanted to mention the **234 character** length, as you'll remember that I explained earlier, that there is a **250 character limit per line**, and I am bearing down on that limitation. However, you can see that quite a bit of expression can be written within that 250 character limitation. You must always keep in mind the limitations imposed upon you by the Lookup.ini.

Having said all of that, I have to continue with our comparison list. What if none of the Lookup variables had matched? What should GoldMine do then? GoldMine would fail all tests, and in doing so, with no other provision, GoldMine would enter a blank into the field. However, I have foreseen this possibility, and told GoldMine that if all of the Lookup variables fail, then to restuff what you believed to be the last name originally, back into the **LastName** field. I do that with my **Otherwise** clause.

```
Otherwise = &LastName
```

I use the GoldMine macro, **&LastName** (refer to Apendix B), to let GoldMine reprocess the **Contact** field, using its own expression, to produce the characters to be pushed into the **LastName** field.

Finally, because the contents of the **Contact** field may change through the course of time, I must assume that a value was previously contained in the **LastName** field, and, in doing so, I must instruct GoldMine to overwrite that field with the newer information. I do that with the statement that you will, by now, recognize as:

```
Overwrite = 1
```

You must remember that even when a new record is created, that as long as there is something in the **Contact** field, there will always be something in the **LastName** field, and that you must account for that fact in your Lookup.ini instruction set.

Running External Applications

Next, I will examine the use of a Lookup.ini for the running of an external application. One typical example of this need is caused by the limitations of the Lookup.ini itself. One of my clients had territories assigned by zip codes. A Lookup.ini to handle the stuffing of several fields based upon all of the zip codes in the United States would far exceed the limitations of the Lookup.ini (not necessarily true with todays operating systems). Sometimes, in these cases, I have tried to combine zip codes into groups to reduce the size of the Lookup.ini. In most cases, however, this has failed. I could easily create an external application that looks through a zip code database for the proper information, and then stuffs that information back into the appropriate GoldMine fields. Although I don't go into the details of the inner workings of the external application in this book, I must know how to execute that application once it has been created should a field change. Additionally, as I'll explain later, you could have an application run when a new record is created, or when a record is edited in any of the main GoldMine tables. Let's look at the case where an application needs to be run when the contents of the **Contact1->Key1** field has changed.

Note

You may include an AutoUpdate action before running your external application.

Tip

You must include the path to the executable if the executable is not in the windows default search path.

```
[AutoUpdate]
  Key1 = Key1

[Key1]
  Run = C:\AnyPath\ZipCode.exe
  RunFlags = 2
```

This is the piece of code that should allow me to do that. I am watching the **Key1** field for a change, and should any change occur, I am instructing GoldMine to process the instruction set for the **Key1** field.

My first statement under the **Key1** instruction set is to tell GoldMine which application to run. I do this with the **Run** statement. GoldMine requires that I set flags to tell it under which conditions the **Run** statement is acceptable to execute. Obviously it is to be run if the **Key1** field changes, but you must consider that as an **.and.** inclusion. That means that not only must the **Key1** field have changed, but the conditions specified by the **RunFlags** statement must be met as well.

The RunFlags statement value is the sum of the values for three possible conditions. These conditions are:

```
Run when the field's lookup value is found          =    1
Run when the field is updated via the [AutoUpdate] section =  2
Run when the field is updated via Automatic Processes    =    4
```

As I do not have any Lookup values to find, and I do not want the application to run during our running of the Automated Processes, I have chosen to use **2** as the value in our **RunFlags** statement. Should you have a Lookup list, and you desire to have this executable run during your running of the Automated Processes, then you would have to use the value of **7** which is the sum of the three possible values.

In this example I will ask the Lookup.ini to watch for a change in the **Key1** field, and with that change, if the value appears in the lookup list of items to run an application. If the value is not included in the lookup list of values, then do not run the application.

```
[AutoUpdate]
  Key1 = Key1

[Key1]

  Lookup1 = upper(trim(Contact1->Key1))

  DD = Dave Dunlap
  DJ = DJ Hunt
  Overwrite = 1

  Run = Calc.exe
  RunFlags = 3
```

With **RunFlags** set to **3**, when a lookup value is found in the list (in this case **DD** or **DJ**), the field is updated accordingly, and then, after the field update, the **Calc** application is run automatically. Placing any other value in the **Key1** field, and absolutely nothing will happen. The field will not be updated, nor will the external application be run.

Now we need to look at the other method for running external applications using the Lookup.ini. As I stated previously, when a new record is created, **[OnNewRun]**, or when a record is edited, **[OnEditRun]** in any of the main tables, you may have a need to run an external application. These two sections of the Lookup.ini are independent of the **[AutoUpdate]** section, and they are neither controlled by it, nor react to it. Their statements are tested for validity at all times, and should a condition be met, then the associated executable will be launched.

```
[OnNewRun]

  Cal-S = SaleCApp.exe
  Cal-C = CallCApp.exe
  Cal = CalApp.exe

  ContHist-S = SaleHApp.exe
  ContHist-CI = InCallHApp.exe
  ContHist = HistApp.exe

  ContSupp-P = DetailApp.exe
  Contact1 = NewContact.exe

  Otherwise = AnyOldApp.exe
  AppendRecNo = 1
  DisableFromAP = 1
```

The sample above is similar to the one that is supplied by GoldMine. The syntax for the **[OnNewRun]** and **[OnEditRun]** is identical with the exception that the former applies only to newly created records of the specified type, while the latter applies only to those records of a specified type that have been edited.

In both cases you must supply the section header. In the sample case I used **[OnNewRun]**, but I could have just as easily used **[OnEditRun]**. In these sections, one defines the left hand side of the equation with the table and/or the table record type that would required to be added or edited for the application on the right hand side of the equation to be executed.

Upon close examination of the sample, you will notice that the tables are identified only by their names. Consequently, any time a new record is added or edited in the **Contact1**, **Contact2**, **ContSupp**, **ContHist**, or the **Cal** tables, the associated executable will be launched. You must use the table names as they have been identified here.

Alternatively, you may launch an executable based on the addition of, or the editing of a specific type of record in certain tables. Let's look at a couple of the lines.

```
Cal-S = SaleCalcApp.exe
Cal-C = CallCalcApp.exe
```

To identify record types, you must use the table name, a dash, and then the information contained in the **RecType** field of the table (refer to The Tables chapter). In the first statement, above, I use the **RecType** of **S** for sales record. Therefore, the addition of, or the editing of a **Forecasted Sale** within GoldMine will trigger the launching of the **SaleCalcApp.exe** executable. Similarly, in the second statement, I use the **RecType** of **C** for a **Call** record. If a call is scheduled in, or edited from the calendar then the **CallCalcApp.exe** executable will be launched.

Tip

If the executable does not reside in the same directory as the Lookup.ini, you must disclose the full path to the executable in your statement.

Here are the RecTypes that can be used:

Cal		ContHist		ContSupp	
A	Appointment	A	Appointment	C	Additional Contact
C	Call Back	CC	Call Back	L	Linked Document
D	To-do Action	CI	Incoming Call	O	Organizational Chart
M	Message	CM	Returned Message	P	Detail Record
O	Other	CO	Outgoing Call	R	Referral Record
S	Forecast Sale	D	To-do Action		
T	Next Action	L	Letter		
		M	Message		
		O	Other		
		S	Sale		
		T	Next Action		

Note

*In the **ContHist** table, only the first **2** positions of the **RecType** field are monitored.*

There are no **RecTypes** contained in either the **Contact1** or the **Contact2** table.

As with all of the other instruction sets, GoldMine permits the usage of the **Otherwise** clause. In my example I used:

```
Otherwise = AnyOldApp.exe
```

While just below that statement, there were two special instructions for this section. They were:

```
AppendRecNo = 1
DisableFromAP = 1
```

Note

*In both statements, a **1** sets the switch to **True**, while a **0**, or no statement at all, sets the switch to **False**.*

The first, **AppendRecNo**, instructs GoldMine to append the number of the record at the current pointer position in the table as a parameter to the launching executable. In the **Cal-C** statement, if the record pointer were at **RecID BRQ1DSJ#(QQ%R#Y** in the calendar table, then the Lookup.ini launching instruction would look similar to:

```
CallCApp.exe BRQ1DSJ#(QQ%R#Y
```

Note

*You cannot believe the GoldMine Premium Help files on this, as they are way outdated. Against SQL tables, where there are no record numbers, the **RecID** is passed in lieu of the **RecNo**.*

By using a parameter such as I have described, your external application may make use of the GoldMine Dynamic Data Exchange (**DDE**) capability to locate the appropriate record in the **Cal** table, and take action against it and/or extract the information from it. In fact, passing this parameter is the only way for the external application to know where the record pointer is located. In a previous statement, I mentioned that I wrote an application that would populate fields based on the zip code. That application had to be instructed as to which record in the **Contact1** table to add this information into. This is the very method that I employed for the launching of that application.

The final statement employed the switch **DisableFromAP**. This instruction tells GoldMine to disable all of the options in this section if the newly added record, or edited record resulted as consequence of the execution of a GoldMine Automated Process.

Playing Macros

GoldMine allows the users to create keyboard macros, and these macros could be played through the appropriate use of your Lookup.ini. The keyboard macros could be played either through an instruction set or through the **[OnNewRun]** or the **[OnEditRun]** sections of the Lookup.ini. This adds yet another new level of functionality to your Lookup.ini. Let's look at a simple Lookup.ini that makes use of the **PlayMacro(<MacroNumber>)** function.

```
[AutoUpdate]
  Key1 = Key1
```

Tip

Keyboard macros are user specific. If you plan to play a keyboard macro through the use of the Lookup.ini, then you must assure that the macro number that is being played is identical for each and every user in your GoldMine system.

Tip

Keyboard macros, which move from the current locked record to another record, may prevent the calling function from completing properly. Try to avoid situations through the Lookup.ini where the playing keyboard macro changes to another contact record.

Note

Keyboard macros tend to falter after extended periods of use for some unknown reasons. You are advised to periodically refresh the users Keyboard Macros.

```
[Key1]
  Lookup1 = Key1

  West Coast = &PlayMacro(841)
  East Coast = &PlayMacro(840)

[OnNewRun]
  Contact1 = &PlayMacro(835)
```

I first ask GoldMine to watch the **Key1** field for any change, and, if there is a change, to follow the instruction set for **[Key1]**. I then make use of the **Lookup1** variable to look at the current value of the **Key1** field, and to compare that to my list of values. If there is a match against **West Coast**, then GoldMine is instructed to play the keyboard macro number **841** which belongs to the currently logged in GoldMine UserID.

You probably would benefit more if I put this into a real scenario. Suppose that every time a new customer record was added to GoldMine, that you wanted to have that person scheduled for a call back. You would first record your keyboard macro to schedule a call back activity, and save it to one of the supplied GoldMine icons. The first icon in the column of icons is number **800**, while the next is **801**, and so forth on down the column. As you can see above, it is important that you know the number of the keyboard macro to be played. I could now employ the **[OnNewRun]** section of the Lookup.ini file to watch for all newly created records in the **Contact1** table, and as each is created to play the appropriate macro.

If you are not adept at writing external applications, you can usually circumvent that need by creating keyboard macros that will perform most of your actions. You can then have those keyboard macros played through the use of your Lookup.ini. Make sure that you read the sidebar Tips as they contain some important additional information pertaining to this section.

Color Coding Calendar Activities

Your organization may want to standardize their color coding schema for calendar activities. Why it is a part of the Lookup.ini, I do not know, but it is, hence, I will discuss it now. The Lookup.ini can have another independent section called **[CalClrCode]** which identifies your corporate color schema for calendar activities. Here is a typical example of its usage.

```
[CalClrCode]
  A = 3
  C = 1
  T = 4
  M = 0
  O = 2
```

On the left hand side of the equation, you identify the activity that is to receive the specified color that is contained in the coded number in the right hand side of the equation. Below I have listed the coding for each side of the equation.

Left Hand Side	Right Hand Side		
A = Appointments	0 = Bright Blue	5 = Bright Yellow	10 = Green
C = Calls	1 = Bright Purple	6 = Cyan	11 = Yellow
T = Next Actions	2 = Bright Red	7 = White	12 = Blue
M = Messages	3 = Bright Cyan	8 = Gray	13 = Purple
O = Other Actions	4 = Bright Green	9 = Red	14 = Dark Grey

One can clearly see that I have asked for my appointments to be colored bright cyan, while my calls will be colored bright purple on the GoldMine graphical calendar interface. This schema, or the one that you may develop for your organization, will provide another level of consistency in your organization. No matter which users graphical calendar that you are examining, the calendars will be consistently color coded such that everyone will know that it is an appointment if it is bright cyan for instance. A caveat here: Users may opt to change the color when Scheduling or Modifying a Scheduled Activity. Consistency in application usage makes for a better all around use of the GoldMine application itself. Strive for consistency wherever and whenever you can achieve it in your corporate development of the GoldMine product.

But wait, we are not finished. FrontRange has now provided a more granular approach to the coporate color coding schema capabilities.

```
[CalClrCode]
  A = 4
  A-TRG = 5
  C = 1
  C-FUP = 2
  O = 2
```

That's correct, we can now begin to further the **ActvCode** level. For instance: The **A-TRG = 5** statement is stating that if this scheduled **Activity:** is that of an Appointment type, and if the **Code:** is **TRG** then the color coding for this activity should be **5** or **Bight Yellow**. Actually, this capability existed before GoldMine Premium 8.5.1.12, which is the version that I'm currently writing this book against, however, one of my readers brought this to my attention so that I could include it in future books such as now. You readers are sometimes my best resources. Either way this is way cool, because now you can use corporate color coding to segregate those special activities like **Forecast Sales** of **PRO** (Prospects).

Generating Your Own Unique Identifier

Using your Lookup.ini, you could generate your own unique identifier for a newly created record, and have that identifier populate one of your fields.

Why would I want a unique identifier on top of the AccountNo and RecID which are already unique?

GoldMines AccountNo, and RecID employee special higher level characters that are not recognized by all systems. The unique identifier that I am generating employees only standard characters and numbers, and can be more easily used when Importing into GoldMine while trying to match existing GoldMine records for updating.

To accomplish this, I will make use of the **Counter()** function. There is one thing that is important to understand before I begin this exercise. The **Counter()** function stores the number that it produces in the GoldMine **Lookup** table.

Why is this important?

That is also a very good question for you to have asked. If you are synchronizing your GoldMine data, then this number, as it resides in the **Lookup** table at the time of the synchronization, will synchronize out to those remote users. This means, in using this number alone, you are very likely going to generate duplicate numbers if your records are not created from a single location. Someone could be retrieving the next sequential number on the server while a remote user is retrieving that same sequential number on their remote GoldMine. Therefore, you must never use the **Counter()** function by itself to generate a unique number when there is any chance of synchronization within your organization. What I propose, knowing that the user login name is unique within GoldMine, is to append the user login name to a padded form of the counter. This, then, will always generate a unique number for each record even in a synchronization scenario. Here is the code that I propose that you employee to achieve this unique number.

Tip

*As you are generating a unique ID, you may wish to protect the field from changes. You should consider setting the **Update rights:** field, for this field to **MASTER**, or at least to a group of users having **Master Rights**.*

```
[AUTOUPDATE]
  NewRecord =  Key5

[KEY5]
  Otherwise = &&UserName+padl(ltrim(str(counter([AcctNo],1))), 8, [0])
```

That is quite a mouthful, wouldn't you say? Obviously, I am watching for a newly created record, and once GoldMine realizes that a new record has been created, to process the instruction set for the **Key5** field. Simple, now that you know the process, isn't it? I already know that the leading **&** in the **Otherwise** statement lets GoldMine know to evaluate the remainder of the string as an expression. The first part of our equation is **&UserName**, which is a GoldMine macro function (refer to Appendix B). This macro function extracts the trimmed **UserID** login name of the active user as a string. I say trimmed, because, as you are aware, the **UserID** field can contain up to eight characters. The **&UserName** macro only extracts the name from this field, and not any of the additional spaces that may be present.

To the **UserID** login name, I am then appending, working from the inner most parenthesis outward, the **Counter()** function, the **Str()** function, the **LTrim()** function, and finally the **PadL()** function. All of these functions are covered in more detail in Appendix A at the end of this book.

However, in pseudo code, I am:

- retrieving the next sequential number for the variable **AcctNo**, **Counter()**
- converting that number to a string, **Str()**
- trimming off any spaces from the left side of the string, **LTrim()**
- padding the left side of the string to eight characters with the character **0**, **PadL()**

Record Typing (Another Approach)

GoldMine Premium contains the **Record Typing** concept, as I had discussed in Chapter 4. As the GoldMine Administrator, it is important that you take this into consideration when you are developing your Lookup.ini.

In Chapter 4, I had discussed employing the **Contact1->Key1** field for **Record Typing**. I had discussed that this field could identify the underlying record as being either a **Buyer**, **Seller**, **Agent**, or **Property** record. I then discussed using one field to hold different information based upon the record type. Let's say, for this example, that I am going to use the **Contact2->UserDef01** field to hold the

availability date if this is a record type of **Property**. While a record type of **Buyer**, or **Seller**, this same field could contain a **Status**. I am also going to consider in what state the **Agent** may be located, and supply the agents name associated with that state. Let's see how this might look in the Lookup.ini.

```
[AutoUpdate]
  Key1 = UserDef01

[UserDef01]

  Lookup1 = upper(trim(Contact1->Key1))
  Lookup2 = upper(trim(Contact1->Key1))+upper(trim(Contact1->State))

  PROPERTY = &dtoc(date())
  BUYER = Ready to Purchase
  SELLER = Available for Sale
  AGENTMA = DJ Hunt
  AGENTNH = Davey Crackett
  AGENTME = Waldo Fairchild
```

I am simply reminding you that **Record Typing** throws a whole new monkey wrench into the Lookup.ini development environment. As the GoldMine Administrator, you must take this into consideration when developing your Lookup.ini.

Currency Formatting

Many users have asked to have the ability to format a field in a currency format. Many financial institutions, using GoldMine, have asked me for this capability. Here is a little Lookup.ini code that I have developed to accomplish this. To use this code you need a character based field, so I added the field:

```
uDlrFormat, C, 15
```

Here is the Lookup.ini code to keep the Contact2.uDlrFormat field formatted properly:

```
[AutoUpdate]
  uDlrFormat = uDlrFormat

[uDlrFormat]
  Lookup1 = alltrim(str(len(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)))))
      +iif(“.” $ Contact2->uDlrFormat, “T”, “F”)

  2F = &”$”+alltrim(Contact2->uDlrFormat)+”.00”
  3F = &”$”+alltrim(Contact2->uDlrFormat)+”.00”
  4F = &”$”+left(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)),1)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 2, 3)+”.00”
 5F = &”$”+left(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 2)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 3, 3)+”.00”
  6F = &”$”+left(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 3)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 4, 3)+”.00”
  7F = &”$”+left(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 1)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 2, 3)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 5, 3)+”.00”

  5T = &”$”+alltrim(Contact2->uDlrFormat)
  6T = &”$”+alltrim(Contact2->uDlrFormat)
  7T =&”$”+left(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 1)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 2, 6)
  8T =&”$”+left(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 2)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 3, 6)
  9T =&”$”+left(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 3)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 4, 6)
 10T =&”$”+left(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 1)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 2, 3)+”,”
      +substr(trim(strtran(strtran(Contact2->uDlrFormat, “,”,””),”$”,””)), 5, 6)

Otherwise = &Contact2->uDlrFormat
Overwrite = 1
```

Note

Remember that each line is a continuous line of code. The lines are wrapped, and indented here for presentation and readability only.

The premiss behind this code is simple. The Lookup.ini is watching the **Contact2.uDlrFormat** field for changes, and, when the field changes, is processing the instruction set for that field (itself). All the coding is based on the length of the characters in the field after the dollar sign (**$**) and commas (**,**) have been removed. In the Lookup.ini, I append that length to a **T**rue or **F**alse letter if the value contains a decimal.

Once I have populated the Lookup1 variable with a value, I assign instruction sets based on that value. For instance, a Lookup1 value of **7T** means that the value in the field, without a dollar sign or commas, is **7** characters long, and the **T** means that one of those seven characters is a decimal. Formatting this number becomes easy. I add back in the $ sign, and add to that the 1st character of the field, and a comma. To that, I then simply add the rest of the characters.

Lookup.ini Razzle - Dazzle

A number entered as **1234.56**, would be processed, and returned back to the field as **$1,234.56** as would this **$1234.56** , or this **1,234.56** if it were entered into that same field. Perfect. This is exactly what your client was looking to achieve.

I have already commented on how important consistency of data in GoldMine is, and how difficult that is to achieve. This section of this chapter will cover a Lookup.ini that I had created for my Financial Institution type of GoldMine usage, however, it contains many examples of the different ways in which a Lookup.ini can be utilized.

Here are the prerequisite fields that are needed, in addition to the GoldMine default fields, so that this Lookup.ini will function properly:

```
UserDef01, C 40
uALabelA, C, 40
uALabelB, C, 40
UserDef02, C, 40
UserDef03, C, 40
uNNCl, C, 20
uNNMa, C, 20
uADearA, C, 20
uADearB, C, 20
uAAddress1, C, 40
uAAddress2, C, 40
uAAddress3, C, 40
uACity, C, 30
uAState, C, 20
uAZip, C, 10
uCTaxID, C, 12
uMTaxID, C, 12
uSSNCl, C, 12
uSSNMa, C, 12
```

Throughout the Lookup.ini, I will use Comment lines to annotate the functions of the various instruction sets. These Comment lines will be in bold black, and they will be preceded by a semicolon (;).

```
[AutoUpdate]
  NewRecord = UserDef01, Company, LastName, Department, uALabelA, uALabelB, Contact, Dear,
              Secr, UserDef02, UserDef03, uNNCL, uNNMA, uADearA, uADearB
  Contact = LastName, Department, uALabelA, uALabelB, Contact, Dear, Secr, UserDef02, UserDef03,
            uNNCL, uNNMA, uADearA, uADearB
  State = Country, uAState, uACountry
  Company = UserDef01, Company
  Address1 = uAAddress1
  Address2 = uAAddress2
  Address3 = uAAddress3
  City = uACity
  uAState = uACountry
  Zip = uAZip
  uCTaxID = uCTaxID
  uMTaxID = uMTaxID
  uSSNCl = uSSNCl
  uSSNMa = uSSNMa

[COMPANY]

; This instruction set will consistently enter a Company name into the Company field.
; If the Company name is entered as The Financial Institution, this instruction set will convert
; that name to Financial Institution, The
; Any Company name that does not begin with the word The will remain as entered

  Lookup1=iif(upper(left(&Company,4))==[THE ], 'A', 'Z')

  A = &substr(&Company, 5, 40)+[, The]
  Otherwise = &&Company

  Overwrite = 1

[CONTACT]

; Here I am looking for any salutation that may have been entered into the Contact field
; and I am stripping it out.  Notice the order of processing in the AutoUpdate section
; above as it is critical to the proper processing of this Lookup.ini

  Lookup1 = iif(upper(left(&Contact, 10)) == [MR. & MRS.], [A], [Z])
  Lookup2 = iif(upper(left(&Contact, 8)) == [MR & MRS], [C], [Z])
  Lookup3 = iif(upper(left(&Contact, 4)) == [MRS.], [D], [Z])
```

Note

Remember that each line is a continuous line of code. The lines are wrapped, and indented here for presentation and readability only.

```
Lookup4 = iif((upper(left(&Contact, 3)) == [MR.] .or. upper(left(&Contact, 3)) == [MRS] .or.
        upper(left(&Contact, 3)) == [MS.]), [E], [Z])
Lookup5 = iif((upper(left(&Contact, 2)) == [MR] .or. upper(left(&Contact, 2)) == [MS]), [F], [Z])
Lookup6 = iif(upper(left(&Contact, 10)) == [DR. & MRS.], [A], [Z])
Lookup7 = iif(upper(left(&Contact, 8)) == [DR & MRS], [C], [Z])
Lookup8 = iif(upper(left(&Contact, 3)) == [DR.], [E], [Z])
Lookup9 = iif(upper(left(&Contact, 2)) == [DR], [F], [Z])

A = &substr(&Contact, 12, 40)
B = &substr(&Contact, 14, 40)
C = &substr(&Contact, 10, 40)
D = &substr(&Contact, 6, 40)
E = &substr(&Contact, 5, 40)
F = &substr(&Contact, 4, 40)
Otherwise = &&Contact

Overwrite = 1

[COUNTRY]

; Here I am automatically determining the Country field base on the information entered into
; the State field.

Lookup1 = iif(trim(&State) $ "MA NY CT VT NH ME RI VA NC TX MI MO KS CA PA FL ND SD HI AL
        KY ID WI DC", "A", "Z")
Lookup2 = iif(trim(&State) $ "UT NM AZ NV OH IL OR WA OK WV SC DE CO MD GA MN IN NJ IA NE
        TN LA AR MS", "A", "Z")
Lookup3 = iif(trim(&State) $ "ON AB NS MB QC", "B", "Z")

A = USA
B = Canada
Otherwise = &&Country

Overwrite = 1

[DEAR]

; Here I am looking for any compound names, and stripping them apart for the Dear field.
; For instance, if the Contact name were entered as DJ & Carol Hunt, this instruction set would
; populate the Dear field with DJ & Carol.

; Pay particular attention to the order of processing in the AutoUpdate section as it is
; critical to the proper processing of this Lookup.ini

Lookup1 = iif([ AND ] $ upper(&Contact), [A], [Z])
Lookup2 = iif([ & ] $ upper(&Contact), [B], [Z])

A = &&FirstName+[ & ]+left(strtran(&Contact, &FirstName+[ and ], []), at([ ], strtran(&Contact,
        &FirstName+[ and ], [])))
B = &&FirstName+[ & ]+left(strtran(&Contact, &FirstName+[ & ], []), at([ ], strtran(&Contact,
        &FirstName+[ & ], [])))
Otherwise = &&FirstName

Overwrite = 1

[DEPARTMENT]

; Here I am using the Department field to hold any Salutation or Contact name Prefix.
; For instance, if the Contact name were entered as Dr. and Mrs. DJ & Carol Hunt, this instruction
; set would populate the Department field with Dr. and Mrs..

; Pay particular attention to the order of processing in the AutoUpdate section as it is
; critical to the proper processing of this Lookup.ini

Lookup1 = iif(upper(left(&Contact, 10)) == [MR. & MRS.], [A], [Z])
Lookup2 = iif(upper(left(&Contact, 8)) == [MR & MRS], [C], [Z])
Lookup3 = iif(upper(left(&Contact, 4)) == [MRS.], [D], [Z])
Lookup4 = iif((upper(left(&Contact, 3)) == [MR.] .or. upper(left(&Contact, 3)) == [MRS] .or.
        upper(left(&Contact, 3)) == [MS.]), [E], [Z])
Lookup5 = iif((upper(left(&Contact, 2)) == [MR] .or. upper(left(&Contact, 2)) == [MS]), [F], [Z])
Lookup6 = iif(upper(left(&Contact, 10)) == [DR. & MRS.], [A], [Z])
Lookup7 = iif(upper(left(&Contact, 8)) == [DR & MRS], [C], [Z])
Lookup8 = iif(upper(left(&Contact, 3)) == [DR.], [E], [Z])
Lookup9 = iif(upper(left(&Contact, 2)) == [DR], [F], [Z])

A = &left(&Contact, 10)
B = &left(&Contact, 12)
C = &left(&Contact, 8)
D = &left(&Contact, 4)
E = &left(&Contact, 3)
```

Note

Remember that each line is a continuous line of code. The lines are wrapped, and indented here for presentation and readability only.

```
  F = &left(&Contact, 2)
  Otherwise = &&Dept

  Overwrite = 1

[LASTNAME]

; Here I am properly populating the LastName field.  This uses the same LastName Instruction
; Set discussed earlier in this chapter.

; Pay particular attention to  the order of processing in the AutoUpdate section as it is
; critical to the proper processing of this Lookup.ini

  Lookup1 = iif(“,” $ Contact, “A”, “Z”)
  Lookup2 = iif((upper(trim(LastName)) == “JR” .or. upper(trim(LastName)) == “JR.”), “B”, “Z”)
  Lookup3 = iif((upper(trim(LastName)) == “SR” .or. upper(trim(LastName)) == “SR.”), “B”, “Z”)
  Lookup4 = iif((upper(trim(LastName)) == “II” .or. upper(trim(LastName)) == “III”), “B”, “Z”)
  Lookup5 = iif((upper(trim(LastName)) == “ESQ” .or. upper(trim(LastName)) == “ESQ.”), “B”, “Z”)
  Lookup6 = iif((upper(trim(LastName)) == “MD” .or. upper(trim(LastName)) == “MD.”), “B”, “Z”)
  Lookup7 = iif((upper(trim(LastName)) == “PHD” .or. upper(trim(LastName)) == “PHD.”), “B”, “Z”)

  A = &alltrim(substr(&Contact, rat(“ “, substr(&Contact, 1, rat(“,”, trim(&Contact))-1))+1,
                rat(“,”, trim(&Contact)) - rat(“ “, substr(&Contact, 1, rat(“,”, trim(&Contact))))-1))
  B = &alltrim(substr(&Contact, rat(“ “, substr(&Contact, 1, rat(“ “, trim(&Contact))-1))+1,
                rat(“ “, trim(&Contact)) - rat(“ “, substr(&Contact, 1, rat(“ “, trim(&Contact))-1))))
  Otherwise = &LastName

  Overwrite = 1

[SECR]

; Here I am using the Secr field to hold Marital Status, and populate it when known.
; For instance, if the Department field contains Mr. & Mrs. the assumption is that they are
; Married.

; Pay particular attention to  the order of processing in the AutoUpdate section as it is
; critical to the proper processing of this Lookup.ini

   Lookup1 = iif((upper(trim(Contact1->Department)) == [MR. & MRS.] .or. upper(trim(Contact1-
                >Department)) == [MR & MRS]), [A], [Z])
   Lookup2 = iif((upper(trim(Contact1->Department)) == [MR. AND MRS.] .or. upper(trim(Contact1-
                >Department)) == [MR AND MRS]), [A], [Z])

  A = Married
  Otherwise = &Contact1->Secr

  Overwrite = 1

[UAADDRESS1]

; Here I am populating an Alternate Address field with the information from Contact1.Address1
; Later this information can be changed if the Contact has, let’s say, a summer address.

  Otherwise = &Address1

[UAADDRESS2]

; Here I am populating an Alternate Address field with the information from Contact1.Address2
; Later this information can be changed if the Contact has, let’s say, a summer address.

  Otherwise = &Address2

[UAADDRESS3]

; Here I am populating an Alternate Address field with the information from Contact1.Address3
; Later this information can be changed if the Contact has, let’s say, a summer address.

  Otherwise = &Address3

[UACITY]

; Here I am populating an Alternate City field with the information from Contact1.City
; Later this information can be changed if the Contact has, let’s say, a summer address.

  Otherwise = &City

[UACOUNTRY]

; Here I am populating an Alternate Country field with the information from Contact1.State
; Later this information can be changed if the Contact has, let’s say, a summer address.
```

Note

Remember that each line is a continuous line of code. The lines are wrapped, and indented here for presentation and readability only.

```
  Lookup1 = iif(trim(&State) $ "MA NY CT VT NH ME RI VA NC TX MI MO KS CA PA FL ND SD HI AL
            KY ID WI DC", "A", "Z")
  Lookup2 = iif(trim(&State) $ "UT NM AZ NV OH IL OR WA OK WV SC DE CO MD GA MN IN NJ IA NE
            TN LA AR MS", "A", "Z")
  Lookup3 = iif(trim(&State) $ "ON AB NS MB QC", "B", "Z")

  A = USA
  B = Canada
  Otherwise = &&Country

[UADEARA]

; Here I am populating an Alternate Dear field with the information from Contact1.Dear
; Later this information can be changed if the Contact has, let's say, a summer address.

  Otherwise = &Dear

[UADEARB]

; Here I am populating a second Alternate Dear field with the information from Contact1.Dear
; Later this information can be changed if the Contact has, let's say, a summer address.

  Otherwise = &Dear

[UALABELA]

; Here I am populating an Alternate Contact field with the information from Contact1.Contact
; Later this information can be changed if you want to have an alternate Contact representation.

  Otherwise = &Contact

[UALABELB]

; Here I am populating another Alternate Contact field with the information from Contact1.Contact
; Later this information can be changed if you want to have an alternate Contact representation.

  Otherwise = &Contact

[UASTATE]

; Here I am populating an Alternate State field with the information from Contact1.State
; Later this information can be changed if the Contact has, let's say, a summer address.

  Otherwise = &State

[UAZIP]

; Here I am populating an Alternate Zip field with the information from Contact1.Zip
; Later this information can be changed if the Contact has, let's say, a summer address.

  Otherwise = &Zip

[uCTaxID]

; Here I am populating a TaxID field for the Client in a proper, and consistent format.
; Let's say that the user enters 061-307-252.  This instruction set would convert that to
; 06-1307252

  Lookup1 = iif(empty(Contact2->uCTaxID), [A], [Z])

  A = &space(0)
  Otherwise = &left(strtran(strtran(Contact2->uCTaxID, [ ], []), [-], []),2)+[ ]+substr(strtran(
              strtran(Contact2->uCTaxID, [ ], []), [-], []),3,11)

  Overwrite = 1

[uMTaxID]

; Here I am populating a TaxID field for the Spouse in a proper and consistent format.
; Let's say that the user enters 061-307-252.  This instruction set would convert that to
; 06-1307252

  Lookup1 = iif(empty(Contact2->uMTaxID), [A], [Z])

  A = &space(0)
  Otherwise = &left(strtran(strtran(Contact2->uMTaxID, [ ], []), [-], []),2)+[-]+substr(strtran(
              strtran(Contact2->uMTaxID, [ ], []), [-], []),3,11)

  Overwrite = 1
```

Note

Remember that each line is a continuous line of code. The lines are wrapped, and indented here for presentation and readability only.

```
[uNNCL]

; Here I am populating a Nick Name field for the Client.  This Instruction Set is using the
; GoldMine macro, &FirstName, to extract the First Name from the Contact1.Contact field.

  Otherwise = &&FirstName

  Overwrite = 1

[uNNMA]

; Here I am populating a Nick Name field for the Spouse or Significant Other.  If the Contact1.Dear ;
field were to contain DJ & Carol, then this Instruction Set would extract Carol from that field.

  Lookup1 = iif([ AND ] $ upper(&Dear), [A], [Z])
  Lookup2 = iif([ & ] $ upper(&Dear), [B], [Z])

  A = &strtran(&Dear, &FirstName+[ and ], [])
  B = &strtran(&Dear, &FirstName+[ & ], [])

  Overwrite = 1

[USERDEF01]

; Here I am populating an Alternate Company field with the information from Contact1.Company

  Otherwise = &Company

  Overwrite = 1

[USERDEF02]

; Here I am populating a Contact field with a true, and consistent format.
; Let's say that the Contact name was entered as DJ & Carol Hunt.  This Instruction Set would
; extract DJ Hunt and place it into this Contact field.

  Lookup1 = iif(([ AND ] $ upper(&Dear) .or. [ & ] $ upper(&Dear)), [A], [Z])

  A = &&FirstName+[ ]+Contact1.LastName
  Otherwise = &&Contact

  Overwrite = 1

[USERDEF03]

; Here I am populating a Spouse or Significant Other field with a true, and consistent format.
; Let's say that the Contact name was entered as DJ & Carol Hunt.  This Instruction Set would
; extract Carol Hunt, and place it into this Spousal field.

  Lookup1 = iif([ AND ] $ upper(&Dear), [A], [Z])
  Lookup2 = iif([ & ] $ upper(&Dear), [B], [Z])

  A = &strtran(&Dear, &FirstName+[ and ], [])+[ ]+&LastName
  B = &strtran(&Dear, &FirstName+[ & ], [])+[ ]+&LastName
  Otherwise = &space(0)

  Overwrite = 1

[uSSNCl]

; Here I am populating a SSN field for the Client in a proper, and consistent format.
; Let's say that the user enters 061366738.  This instruction set would convert that to
; 061-36-6738

  Lookup1 = iif(empty(Contact2->uSSNCl), [A], [Z])

  A = &space(0)
Otherwise=&left(strtran(strtran(Contact2->uSSNCl,[],[]),[-],[]),3)+[-]+substr(strtran(strtran(Contact2-
                >uSSNCl, [ ], []), [-], []),4,2)+[-]+right(strtran(strtran(Contact2->uSSNCl, [ ], []), [-], []),4)

  Overwrite = 1

[uSSNMa]

; Here I am populating a SSN field for the Spouse or Significant Other in a proper, and consistent
; format.  Let's say that the user enters 061366738.  This instruction set would convert that to
; 061-36-6738

  Lookup1 = iif(empty(Contact2->uSSNMa), [A], [Z])
```

Note

Remember that each line is a continuous line of code. The lines are wrapped, and indented here for presentation and readability only.

```
A = &space(0)
Otherwise = &left(strtran(strtran(Contact2->uSSNMa, [ ], []), [-], []),3)+[-]+substr(strtran(strtran(
            Contact2->uSSNMa, [ ], []), [-], []),4,2)+[-]+right(strtran(strtran(Contact2->uSSNMa,
            [ ], []), [-], []),4)

Overwrite = 1
```

I hope, from this example, that you can see how really, and truly powerful the Lookup.ini actual could be. I'm sure that if you can think it, that you can also do it through the use of your Lookup.ini.

GMTray

There has been another tool available for consistently formatting data within GoldMine fields for some years now, and it is called **GMTray**. John Stillman, when he had nothing better to do, designed this little ditty as an example for us developers, but it remains today to be a valuable formatting tool. Additionally, it has the added capability of Starting/Stopping GoldMine at predetermined times on specific days.

Note

GMTray is supplied to you in the eBook download as a zipped file. I would suggest that you unzip these files to a local folder on every Workstations C.:\ drive. Possibly: ***C:\GoldMine\GMTray...***

You should then drag ***C:\GoldMine\GMTray\GMTray\obj\Release\GMTray.exe*** *to the Startup folder on that Workstation.*

Now you can either start GMTray.exe by hand or reboot the Workstation, and it will start automatically.

Unfortunately, there is no common shared folder, and GMTray will need to be configured the same for each Workstation to enjoy corporate consistency.

Possibly you have GMTray set up, and working from your Workstations. If so, it is time to configure GMTray by right clicking on the GoldMine treasure chest in the Workstation System Tray (lower right-hand corner of Windows), and selecting **Configure** from the local menu which will bring up the screen shot shown here in Figure 6- 2.

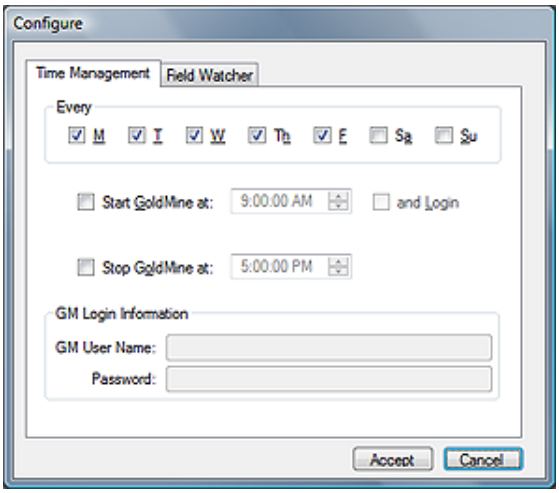


Figure 6-2

The default tab is **Time Management**, and it is from here that one may schedule the starting and stopping of GoldMine for Workstations that are always left running. This was particularly important in the old dBase versions of GoldMine, however, it remains a nice feature even for today's GoldMine. Why leave applications running unnecessarily? The first frame, **Every**, contains simple checkboxes for the days of the week that you wish to have GMTray Start/Stop the Workstations GoldMine. Simply check those days that you wish GMTray Start/Stop GoldMine, and uncheck those days where you do not wish any activity from GMTray.

Naturally, no matter how many days that you have selected in the **Every** frame, nothing will happen unless you have completed the rest of the information in the **Time Management** tab. The first option that is of concern to us is whether to ☐ **Start GoldMine at: 9:00:00 AM**. Obviously, this is the time at which you wish to have GMTray run your GoldMine application from the Workstation. Selecting this alone will cause GoldMine to start up to the splash screen, and sit there waiting for a user to login. Alternatively, they may choose the option ☐ **and Login** which will utilize the **GM Login Information** supplied in that frame to login to GoldMine directly bypassing the splash screen.

Well, once GoldMine is started, and on Workstations that are left running continuously, one may wish to shut down the application as well. To do so, one would have to select to ☐ **Stop GoldMine at: 5:00:00 PM**. For this book, I have chosen to use the default Start/Stop times, however, once selected these times become enabled and the user may address them as is appropriate to their schedule.

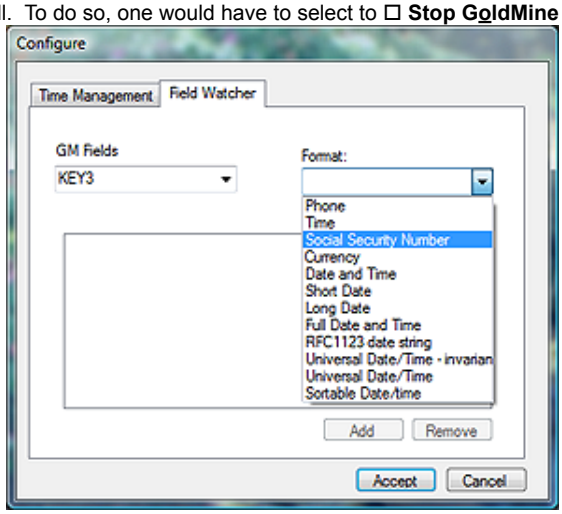


Figure 6-3

Lastly, on the **Time Management** tab, if one wishes GMTray to log the Workstation directly into GoldMine, one must supply the **GM User Name:** and the **Password:** to be utilized for the login in the **GM Login Information** frame.

Next, on the **Field Watcher** tab, we have the ability to set up GMTray to watch certain fields within GoldMine for a change, and when changed, to format the field as specified. One would select field to be watched under **GM Fields**,

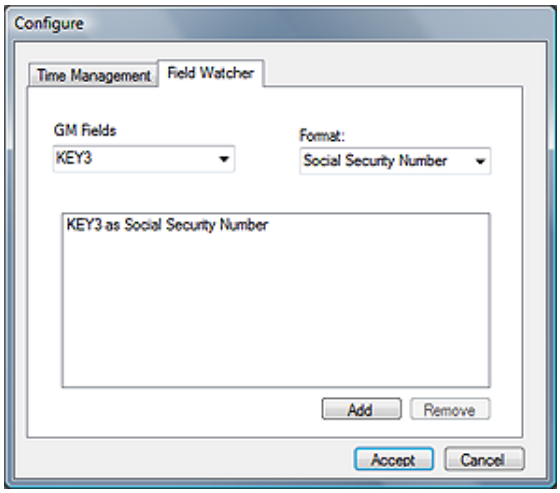


Figure 6-4

and the formatting to take place under the **Format:** field. Once set, one would click on the **Add** button to send it to the list of watched fields.

As you can see here in Figure 6-4, I have set my **Key3** field format as a **Social Security Number** format. Hence, with GMTray running in the System Tray, upon entering **AB123456789** into the Key3 field, GMTray would, and did, convert the value to **123-45-6789**.

As shown in Figure 6-3 on the previous page, there are many other formatting conditions that can be assigned to any field. GMTray eliminates the need for using the Lookup.ini to simply format fields, and, as a bonus, allows you to Start & Stop GoldMine automatically on predetermined days and at predetermined times. For a tool that John developed to show off the API functionality to developers, this is actually a very practical end user tool, and, best of all, its free.