Software Tool House Inc.

Meta-Update

User's Guide

Release 5.56
Updated: 2015-Sep-30

Preface

Audience

This document is intended for Remedy ARS Administrators and developers.

It is expected that the reader will have knowledge of the Remedy ARS system and be familiar with workflow development. It would behove the reader to be familiar with his ARS server's platform and scripting tools.

Limitation of Liability

This program is provided "as-is". We are in no way liable for any losses arising from your use of this program, the sample scripts, or the documentation. It is your responsibility to evaluate this program. It is your responsibility to backup and protect your data. It is your responsibility to evaluate your use of this program for any particular purpose.

This manual does not represent a commitment to maintain any syntax or operation, nor is it warranted to be complete or accurate.

Copyrights

Updates

This program and this manual may change from time to time. The latest version is available at our web site: www.softwaretoolhouse.com.

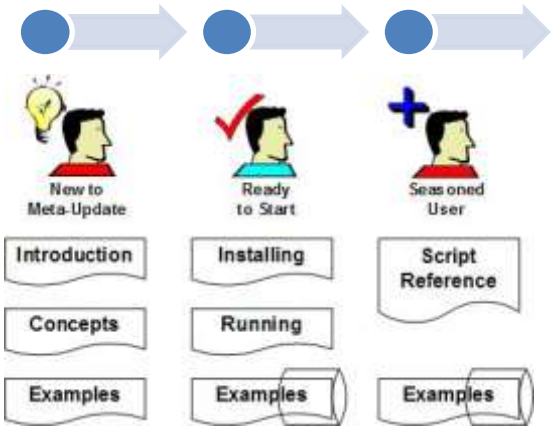
Comments

Your comments are welcome! Please see: www.softwaretoolhouse.com/support and click **Comments**, or email us at support@softwaretoolhouse.com. We look forward to hearing from you!

Organisation

This document is divided into a sequence of sections.

The diagram to the right outlines which User's Guide sections we recommend as you use Meta-Update more and more.



Here's an overview of the User's Guide sections:

Introduction

This is a short description of what Meta-Update is and the types of problems you can solve with it. Read it if you are new to Meta-Update, or want to learn what Meta-Update does at a high level. You do not need to be an ARS Administrator or developer to read this section.

Concepts

In this section, you will learn about the Meta-Update commands and facilities. It introduces the Meta-Update scripting language and gives a general understanding of what to do to implement a script. It is required reading until familiar with Meta-Update scripting.

Installing

The installation notes for UNIX and Windows.

Running

Running Meta-Update and interpreting the output. Developing and debugging scripts.

Script Reference

This explains how to code Meta-Update scripts in detail. You will need to refer to this section when you don't remember how a specific setting or option is specified.

Licensing

This explains how the Meta-Update server based licensing works.

Samples

These samples, which are part of the software distribution and available on the web, are another way to learn how to build scripts. These samples perform different functions and have detailed explanations of the Meta-Update scripts used. Bits of these scripts may be copied and tailored to your scripts.

Software Tool House Inc

Table of Contents

Preface 2
- Organisation 3
- Table of Contents 4

Introduction 11
- Data Challenges 12
- Solution Options 13
- Meta-Update: A New Way to Use The API 14

Concepts 17
- Overview 18
- Definitions 25
- References 27
- Assignments 28
- Assignment References 29
- String References 30
- Loads 31
- Types of Sections 32
- Control Sections 33
- Control Section Flowchart 35
- Iteration 36
  - Query 37
  - QuerySql 38
  - File 38
  - Loop 39
  - Until 40
- Output 41
  - Create 41
  - Update 41
  - Output 42
- Launch 43
- Control Section Examples 44
  - Example: Migrate any Table 44

Installing Meta-Update 47
- Expanding The Distribution 48
- Complete Installation 49
- Distribution Contents 50

Running Meta-Update 55
- Run Time Environment 56
- BMC Remedy API Versions 57
- Program Versions 58
- The License Key 59
- Environment variables 60
  - Script Path Environment variable 60
  - API Retry Environment variable 61
  - License Environment Variable 62

- The Command Line 63
  - Switches 64
  - Usage Help Text 65
  - Using Positional Arguments (Deprecated) 66

Program Return Values 67
Program Output 68
Tracing 71
Local Tracing 72
Server Tracing 73
Trace Format 75
Firing from Workflow 77
Developing Scripts 78

Script Debugging 82
What Is Script Debugging 83
Entering Debug Commands 84
Meta-Update Line Numbers 85
About Meta-Update Break Points 86
Debug Commands 87
List 87
List Files 88
BackTrace 88
Print 89
Next 89
Continue 90
Quit 90
Break 91

Script Reference 95
Script File: General Format 96
Including Other Script Files 98
Section Types 99
[Main] Section 100
Read Server Sections 107
Control Sections 108
About Control Sections 108
Keywords and Statements 109
Load Statements 117
Query Statements 118
QuerySql Statement 122
File Statement 124
Loop Statement 125
Create Statement 131
Until Statement 131
Update Statement 133
Output Statement 135
Merge Statement 137
Status Statement 137
Sleep Statement 138
Launch Statement 138
IdLog Statement 139

File Sections 142
Field Sections 146
About Fields and Formats 146
Copying Fields from Schemas 147
Field Formats 148
Automatic SQL Select Generation 149
Date Fields 150
Numeric Fields 152
Quotes in Field Values 153

Assignment Reference ... 157
About Assignment Sections ... 158
Using Assignment Sections ... 160
Assignment Targets ... 162
Section Target Types ... 163
Assignments ... 164
Conditional Assignment ... 165
IF Statement ... 167
LookUp Assignments ... 168
Assignment Commands ... 169

Assignment Commands ... 170
Copy Command ... 170
Include Command ... 173
Abort Command ... 174
AttachSave Command ... 174
Msg Command ... 175
Spawn Command ... 175
Reference Command ... 176
Using Regular Expressions ... 191
Using Arithmetic Expressions ... 193
Set Schema Command ... 196
Trace Command ... 197

LookUp Sections ... 199
Overview ... 199
LookUp Types ... 199
Automatic Tags ... 200
Keywords ... 201
Using Files ... 205
Using a Query ... 209
Using an SQL Query ... 210
Caching LookUp Records ... 213
Using different LookUp Lists ... 213

Field Type Notes ... 215
Diary Fields ... 216
Currency Fields ... 217
Numeric Fields ... 218
Enum or Selection Fields ... 219
Date Fields ... 220
Date/Time Fields ... 221
Attachment Fields ... 224

Predefined Reference Tags ... 226
CTL – Meta-Update Information ... 227
Arg – Program Arguments ... 228
ENV – The Environment ... 228
AR_INFO – ARS Server Information ... 228
*RdSvr*_AR_INFO – ARS Server Information ... 228
AR_INFO – Table of Fields and Values ... 229
CTL – Schema Tag ... 236

Licensing ... 240
How It Works ... 241
Specifying The License Key ... 243
Installing the def File ... 245

Samples **255**
AR Schema Report **258**
AR Server Info Report **261**
Ticket Creation Batch Command **264**
Closed Ticket Replicator **268**
Server Delta Copy **274**
ARS Table Backup and Restore **279**

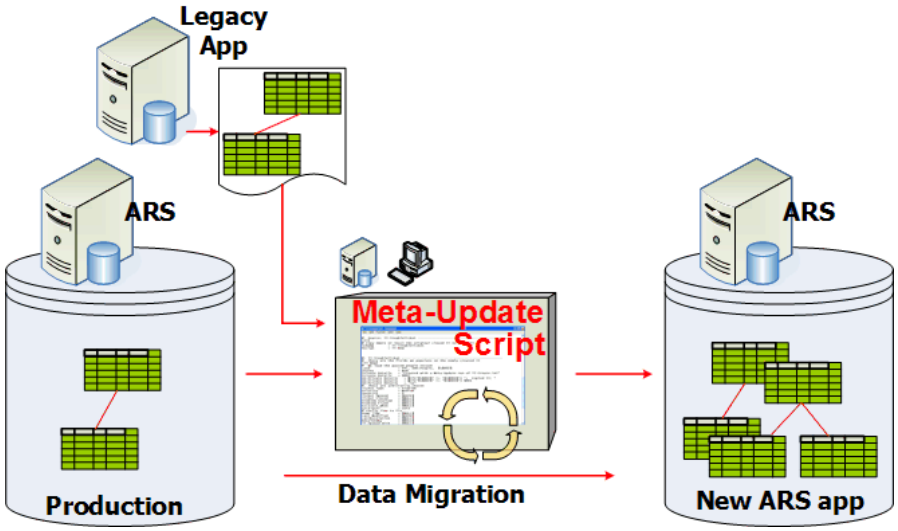
Index **292**

Introduction

Introduction

Thank you for selecting Meta-Update. With Meta-Update, creating repeatable imports, migrations and batch operations on your ARS data is a snap.

Don't bother with the API! Meta-Update provides a quick, robust, reliable, auditable method of harnessing the power of the API without ***any*** programming at all.



Software Tool House Inc

Data Challenges

- Ever had trouble setting up an ARS data migration?
  - From one server version to another?
  - From one release of ITSM to another?
  - From ITSM 6 or 5 or 4 to ITSM 7?
  - From a bespoke ticketing and asset system to another different bespoke application, to an ITSM implementation?

- Ever had trouble importing data into an ARS application?
  - From a series of CSV files representing complex data trees?
  - From CSV files that Excel or the import tool can't handle: containing embedded new-lines, and field values with embedded, undoubled quotes?
  - From CSV files where the query to determine the update record is complex?
  - From CSV files where the target update form changes for each row in the data?
  - From fixed length transactional files / records?

- Ever had trouble getting data transformations right?
  - Assigning the right Status values based upon a different set of incoming values and more complex conditions?
  - Selecting the fields to be updated based upon incoming transaction data, queried data, read data?
  - Setting the values based upon incoming transaction data, queried data, read data?
  - Assigning values to reserved fields like Create Date, and Submitter.

- Ever wanted to adjust, correct, merge, and change the ARS data that you have?
  - Ever needed to combine two clients' foundation data records?
  - Ever wanted to rename or split up support groups?
  - Ever needed to automate the importing of foundation data into the ITSM suite?

- Ever had trouble creating an ARS API program?
  - Ever wasted time talking with a non-ARS programmer?
  - Waited when making assignment or form logic changes for the programming development cycle before seeing the results?

Solution Options

There are four basic choices to solving the types of problems listed above. All are error prone, involved and expensive to develop. Changes to mappings and value interpretations are expensive and slow. In order of performance and efficacy, they are:

1 SQL
Very dangerous to the integrity of the ARS database and workflow. Bypasses all ARS security, safety and workflow mechanisms. Difficult and costly to develop. Requires a specialised resource.

By far the highest performance.

2 API
Given a competent API programmer, this method yields performance approaching the direct SQL option and far better than the next options.

Drawbacks are that ARS API programmers are a rare resource, time between changes and delivery are typically quite long, communications between the ARS developer and the API developer may be difficult.

3 Export, Import, Merge filters
This method should yield reasonable performance though slower than the API.

It is generally a complex ARS development project in and of itself and may rival the API in development expenses.

Much manual effort is needed in the extraction and importing of the data and this is prone to error.

Failures are difficult to track and resolve. Record creation and modification dates are set to current run times and not the historical times required. The same is true of status history and modification users.

There is no facility for parsing through a diary field's entries.

4 User Tool
The fact that this can be done is amazing in and of itself!

Performance alone makes this option not usable. It will have a higher development cost than Export, Import and easily rival or surpass API.

Meta-Update: A New Way to Use The API

With Meta-Update, these types of problems are handled quickly, with ease and confidence!

There is no need for an API programmer or ***any*** programmer at all.

The ARS Administrator / Developer scripts complex functions in the language he already knows in minutes. He fine tunes mappings and assignments and gets his feedback immediately. His runs are fully logged allowing complete resolution and recovery.

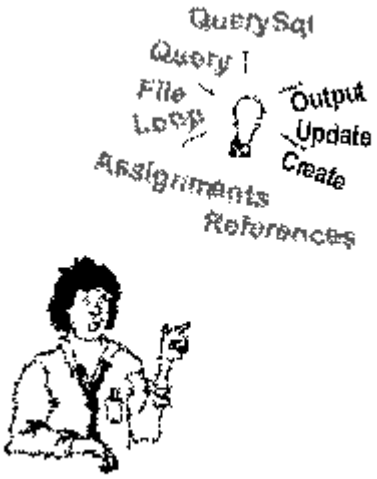
Development efforts for any migration or file import requirements are reduced to at least 1/10th.

That's an order of magnitude savings on the initial development effort compounded by fewer resources required to maintain or enhance scripts from the deployment on.

Compound that development savings with the confidence you get by using Meta-Update:

- The performance is that of the API run on the server or client.
- Jobs complete with "Log and Continue" error processing.
- Errors produce complete resolution and retry information logs.
- Jobs can be broken up in batches and run simultaneously on one or more machines.
- Core fields can be easily assigned on both primary and secondary forms.
- CSV files that fail on the import tool can be handled easily.
- Transactional files can be handled.
- Dates, times, users, status history can be set to any desired value.
- Diary fields' entries can be looped through creating records in other forms.
- All ARS permissions and workflow is respected.

Concepts


QuerySql
Query
File
Loop
Output
Update
Create
Assignments
References

Concepts

Software Tool House Inc

Overview

Meta-Update is an Extract Transform Load (ETL) scripting tool for the BMC Remedy data. Meta-Update uses the BMC Remedy API.

With Meta-Update, you can create any repeatable, programmable import, migration, or, batch job in 1/10th the time it would take to develop a similar function using the ARS API with Perl, Java, or c.

Meta-Update gives an ARS Administrator the power of the ARS API without having to know the API or programming at all!

With Meta-Update, your ARS Administrator can create complex imports and migrations himself, in a language he already speaks, easily and directly.

He tries out his ideas, executes a job, and sees the results. He makes assignment changes and tries and checks again. ***All in minutes!***

Meta-Update gives you a window into the Remedy ARS API. As such, all workflow is fired and all ARS permissions are respected. It does no direct database manipulation at all. It cannot bypass workflow.

Meta-Update Scripting

A Meta-Update run specifies a script file to run. That script can take arguments, load configuration records, load value translate tables, and perform operations on your Remedy data.

A script file is a sectioned INI file similar to Windows and Linux INI files. Data operations and assignments are specified in named sections.

Control sections specify assignment sections where you can assign values to your fields in a simple and feature rich way.

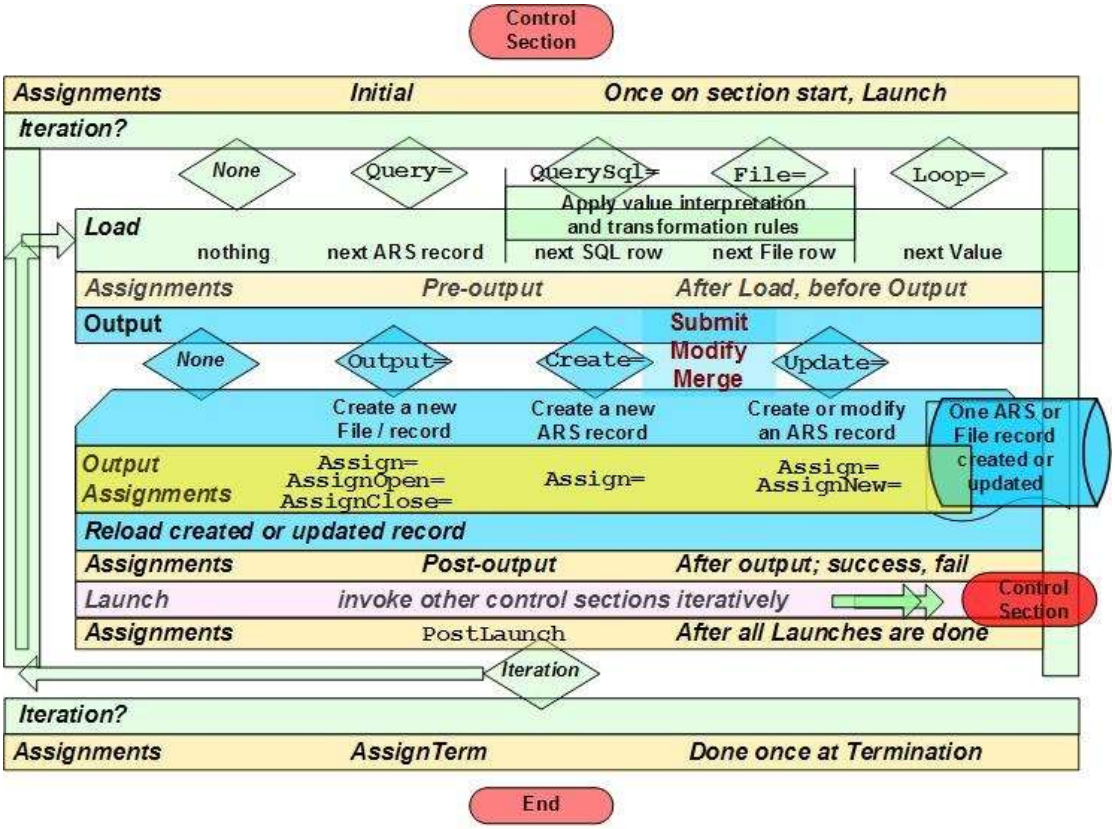
With Meta-Update, you have many different ARS records available that you can use to make your assignments and decisions in your assignments.

```
[UpdCpy]
# Update the Company Name (only) in any TT
# Use SQL to get the list (fields not indexed),
#   then Load and update the ARS record.
QuerySql  =   TT-ID, @na,                          &
              select instanceid from hpd_help_desk    &
              where  company = '$Arg, Cpy$' or        &
              contactcompany = '$Arg, Cpy$ or         &
              supportcompany = '$Arg, Cpy$
LoadQ     =   TT,                                   &
              HPD:Help Desk,                          &
              '179' = "$TT-ID, 1$"
Update    =   TT-Upd,                               &
              HPD:Help Desk,                          &
              '179' = "$TT-ID, 1$"
Merge     =   Yes, NoWorkflow
Update    =   TT-Upd
```

Control Sections

A Meta-Update script comprises a number of linked together "Control Sections" or work elements.

This image summarises the basic flow of a Meta-Update script's Control Section:



Each "Control Section" follows a simple structure as depicted in the image above:

Iteration

A Control Section may choose to ***iterate*** by issuing an ARS Query or processing a CSV file for example.

A Meta-Update control section can iterate through

- an ARS Query,
- an SQL Query,
- an ASCII column file like a CSV,
- a delimited string's values,
- a diary field's entries,.
- while a condition is true
- a set of fields defined in a schema, file, or SQL query,

Or a command section can be run through exactly once.

Within each iteration, Meta-Update loads the next iteration value or record, performs any assignment sections you specify, setting variables, transforming values, and loading additional ARS records or SQL rows.

Output

A section can then optionally perform an output: Creating or Updating an ARS record, appending data to a CSV or text file, or generating the next text file in a pattern of files.

For ARS output, you specify an output form or schema. You can choose to always create a record, or you can specify a query for an update record, and create new records when no records match, and/or update matching records.

You can use Merge if desired, even optionally turning off filters set to fire on Merge.

Of course, you specify your assignments for any outputs and for any script variables.

Launch

After the output record is updated or created, it is reread if needed. This loads a fresh copy after ARS workflow has fired. This then enables subsequent references to use any fields of the updated record.

Finally, after a newly updated or created record is reread, you can launch other, nested control sections. That control section's statements can use the data loaded, queried, created, or updated in the previous sections in all of its substitutions. Because output records are reread, you can use fields like Request IDs, Instance Ids, Group Lists, Diaries, or even attachments

This nesting of sections – which can be made conditional – is what gives Meta-Update its power over the API with significantly simplified development. For each record of an import file, any number of output records in any number of different forms may be created. Each import record can select which form to create a record in.

With a few simple words, your ARS administrator can create chained, nested, data scripts that can be used in automated imports, migrations, or operations, on your ARS data.

Assignments

Throughout each section, different assignments are processed at different events. All assignments can be conditional and many assignment commands are available to allow your administrator exquisite control over the assignment process.

You can programmatically, based on all data in memory, decide to do an update or not, decide which schema and which record to update, decode which fields are to be updated and which values are to be assigned, to which schemas.

The assignments are feature rich. They

- ✓ Are fully conditional supporting a structured if facility.
- ✓ Can use local script variables.
- ✓ Use arithmetic operations;
- ✓ Use pattern matching regular expressions to split up values.
- ✓ Copy matching field ids across two records.
- ✓ Fire external processes on the local processor.
- ✓ Fire special ARS Run Processes on the server, for example, to generate a Guid, or use Business Time.
- ✓ Can use lookup translation facility from internal lists, spreadsheets. SQL, or Remedy forms.
- ✓ Issue Messages and abort processes when detecting data errors.
- ✓ Have full, rich, robust, logging and error tracking facilities.
- ✓ Reference external pattern files to build long text strings using a set of data records across different servers and ARS schemas and ARS SQL rows, or files, or script variables.

All assignments are automatically converted to the right type. ARS keywords can be used. Attachments are no problem. They can be loaded from files or from record references.

References

With the BMC ARS Administrator Tool, you specify references by wrapping a field id or database name in dollar signs. For example:

```
$Request ID$
```

Meta-Update ***extends*** the concept of a reference so that you can refer to many different records at once. A Meta-Update reference has two parts: a Tag that identifies the desired record and a field within that record. For example:

```
$SrcTbl, Request ID$
$TgtTbl, Request ID$
$Arg,    Operation$
```

The Tag portion can refer to
- a Remedy record
- an interpreted SQL row
- an interpreted File record
- a Diary field entry
- a named collection of strings

and the field portion can refer to:
- a field "database" name or field id
- a column or interpreted field
- a file's interpreted field
- the string User, Date, Text
- assigned variables

Value Interpretation

When processing files, SQL queries, regular expression pattern extractions, value interpretations may be applied. For example, in an SQL query, ARS date fields, which are integers, can be interpreted as date fields.

You can request and validate parameters passed as arguments. You can interpret, validate, and transform data from files or SQL queries, pattern extraction Regular Expressions.

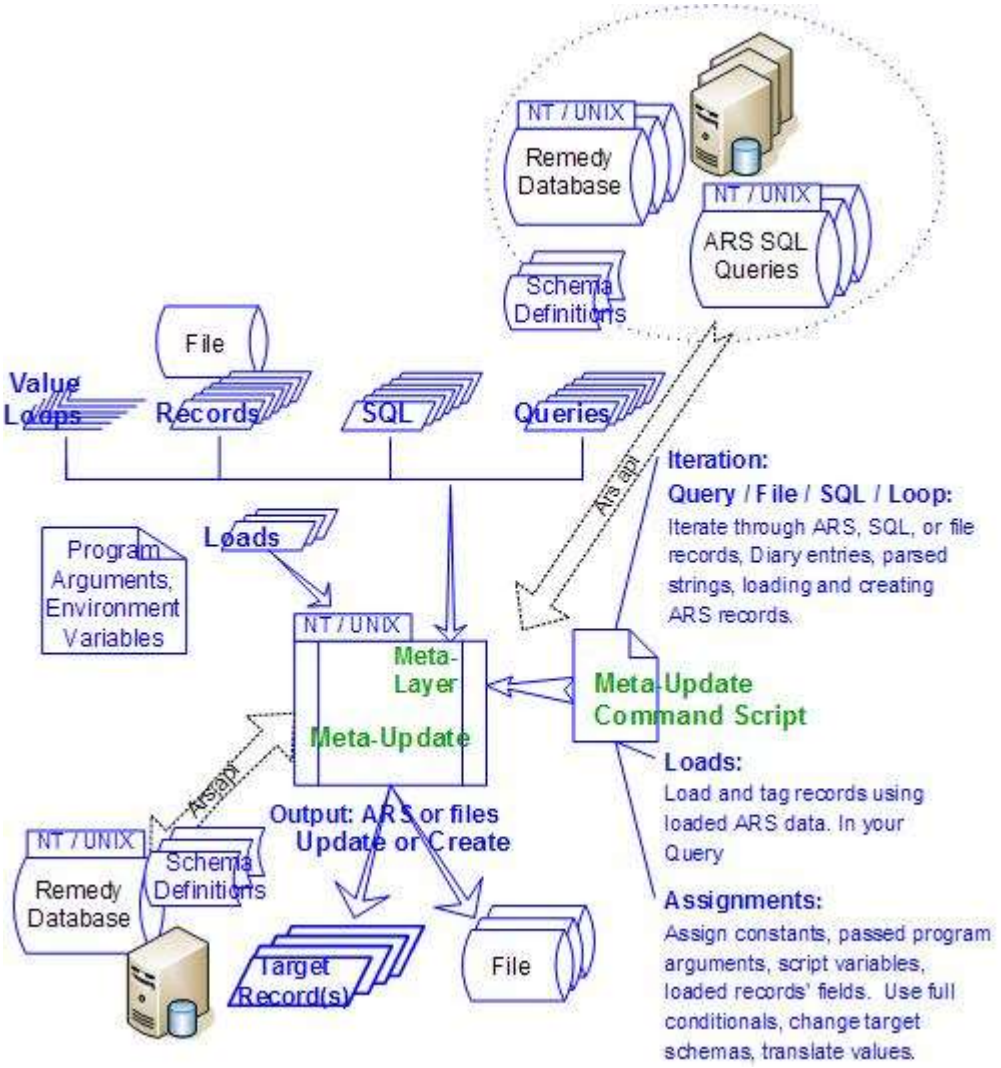
Logging and problem resolution

When processing columnar files, any source records resulting in errors can be written into a new file so that that new file can be then processed by Meta-Update once the data errors are corrected.

Each ARS record read, loaded, created, or updated is optionally written into an "id" log file. You can request and validate parameters passed as arguments. You can interpret, validate, and transform data from files or SQL queries, pattern extraction Regular Expressions.

Meta-Update processes your assignments. You can apply conditionals in the assignments or even in determining to make an update. You can programmatically, based on all data in memory, decode which fields are to be updated and which values are to be assigned.

This image outlines what Meta-Update lets you do:



Meta-Update processes Meta-Update Command Scripts. These are simple ASCII files that resemble Windows 3.1 "INI" files.

In them, with a few simple words and the powerful concept of a "Reference", you tell Meta-Update what to do. You create or update records using Merge Submit, Modify API calls by specifying an update query.

"References" are used throughout a Meta-Update command script.

With the Remedy Administrator, a reference is a field name or id. Meta-Update extends references to include a record identifier, or a "Tag". Within a Meta-Update script that is creating a Ticket, you may have two different people records loaded: one for the requester and one for the requester's manager. The data from both are available when making assignments, running further queries, or in conditional expressions.

A simple Meta-Update command script has a single "command section". In it, there is no Query, SQL Query. or File processed. A Create is coded. So the command will always output a single ARS record. The command script may take a couple of program arguments.

These could be assigned to the new record or used in conditions within the new record assignments.

A Meta-Update command can also be made to iterate, Creating or Updating as many records as the number of times the command section loops. If a Query is coded and it returns 12 records, then 12 records will be output by the Create or Update of the command.

A Meta-Update command can Launch another, different, Meta-Update command. When this is done, the record created or updated in the first command is reloaded, and the new section is run with all References available to it.

So for example, the Launched section could use data in the newly created record to fill in linking information in dependent records.

Definitions

References

References are the key to the power and ease of Meta-Update scripting. In ARS, references are a form's field name or id between dollar signs. Only a single form and record may be so referenced.

Within a Meta-Update, you have many different records in memory at the same time. Each record is identified by a Tag that you chose. So, references comprise two parts:

```
$  Tag,  Field  $
```

Iteration

Meta-Update lets you execute a script command once or iterate though a sequence of "records". Each iteration can load records, make assignments, produce output, and launch other commands.

Query

Allows you to iterate through the results of an ARS query, specified as per the Advanced Query Bar in the BMC User Tool. Of course, the full power of Meta-Update references is available to your query.

QuerySql

Similarly, you can issue a direct SQL query through the ARS API, iterating through the resulting rows, and, naming, interpreting, transforming the resulting columns.

File

Any type of columnar file can be iterated through. Values can be transformed or interpreted. CSV's that Excel or the BMC Import Tool can't handle, are not a problem. CSV's with values having embedded new lines, stray, undoubled quotes, or different delimiters.

Loop

Loops may be based on:

1. diary entries of a diary field, allowing you to create records for each diary entry for example.
2. Delimited strings, for example, looping through a User's Group Permissions field, allowing you to validate and generate individual ITSM People Permission Records
3. A set of defined fields of a schema, for example, looping through all attachment fields to place the attachments on the file system.
4. An arbitrary condition being true – a "while" to process data until a condition is met in any of a set of records
5. The set of schemas making up a join.

Assignments

Assignments are made to create or update ARS records, and in more complex scripts, to use references as variables. Meta-Update has a rich set of assignment commands including a fully nested if – then – else, a Look Up facility, regular expression pattern matching and extraction, an include facility, and other commands.

Output

A Meta-Update command does not have to, but can make, an ARS output, that is, an ARS data record change. A non-iterating command can output a single record. An iterating command can output a single record in each iteration. Output can use either:

- the Merge API (like the BMC Import Tool), firing (or choosing not to fire) all Merge filters, or,
- the Submit and Modify API, like the BMC User Tool minus all Active Links.

Update

Update allows you to specify a query to determine the update record. You can specify different assignment to create a record and to update a record. Once the record is updated, it is reread if it will be needed further in the script.

Create

Create causes a new record to be added to a form. You specify the assignments to be applied. Once the record is created, it is reread if it will be needed further in the script.

Output

You may also choose to create columnar or pattern files as an output of a section. The full power of Meta-Update may be used to load multiple records and use data from all these loaded records in the files you create. Pattern files are text files and can be emails, XML files, or formatted reports. Columnar files can be fixed length or CSVs.

Launch

A Meta-Update command can Launch other commands. For example, a command may iterate through a Query, after each record from the Query is read, and after each output record is reread, it can then Launch another command which can data from the current query record and current output record to query and output more records.

References

With the BMC Visual Studio or the former Remedy Administrator tools, you specify field references by wrapping either the field id or the field's database name in dollar signs.

```
$Status$
$7$
```

Meta-Update can hold many records in memory when it is time to update a Remedy record with assignments. So, a reference comprises two parts, the first specifies the loaded record of an ARS form and the second specifies a field database name or field id from that form.

```
$RecTag,  Status$
$RecTag,  7$
```

The first part is called a "**Tag**". You declare in a Meta-Update statement, such as a **Query=** or **File=**. As each record in the query is processed, the **Tag** that you declare refers to the current record's data. There are also two tags that are set up by Meta-Update.

The second part of a reference is the "field name" as
defined in an ARS form's field's database name,
declared in a **File=**, **QuerySql=**,
as filled by regular expression extracts.
as assigned in a string list,
as implied by field names in the first row of a CSV file,
as set in the environment in the ENV tag,
as set by Meta-Update in the CTL tag.

The "records" that Meta-Update can hold in memory can come from sources other than Remedy forms. Of course, field ids only apply to Remedy forms. In addition to Remedy ARS record references, a Tag may refer to

A CSV or other columnar file's field values
An SQL query column
A named collection of string variables

You can use strings to hold variables allowing you to make complex scripts. There are a set of Meta-Update assigned string variables under the Tag, CTL. Examples are:

```
$CTL,  Pid$
$CTL,  OS$
```

You can also use pattern matching and extraction regular expressions to set up a series of named string variables under a tag corresponding to the extracted values. For example, you could split a status history field into its individual components and generate dated records from those components.

Finally, when assigning references and through the use of assignment commands reference tags may be used as arrays. For example, a configuration file or query may be completely stored in an array of tags and that array may then be processed over and over again.

Assignments

Meta-Update performs field assignments that you code and uses them to update or submit a new record in an ARS form.

```
[cmd1-asg-new]
Status              =      Assigned
Assignment Group+   =      "Web Team"
Description         =      "Automated ticket entry"
Assigned Date       =      $TIMESTAMP$
```

You tell Meta-Update the target form for the field assignments in the `Update=` or `Create=` statement of the command section. Meta-Update then uses the schema to convert the assignments as needed. You specify the assignment sections to process in the command section as well:

```
[Cmd1]
Update        =      HpdUpd, HPD:HelpDesk, '1' = "$Arg, Id$"
AssignNew     =      cmd1-asg-new,       cmd1-asg
Assign                                   cmd1-asg

[Cmd2]
Create        =      HpdNew, HPD:HelpDesk
Assign        =      cmd1-asg-new
```

In the assignment section, the left side of an assignment is a Remedy field name or "id". The right side is a reference to the value you want to assign.

In the above examples, we've assigned three constants and one ARS keyword. Two were text fields; the last, a date field.

The first, Status, was actually assigned the value 1. "Assigned" is validated against the definition of the "Status" field in the target schema, "HPD:HelpDesk". These assignment statements would be equivalent if the default view of form HPD:HelpDesk defined attribute "Assigned" for field "Status" as the second value:

```
Status              =      Assigned
7                   =      Assigned
Status              =      0
7                   =      0
```

The "7" is the ARS field id for the reserved Status field. The value "1" is the normal value associated with "Assigned" in the Attributes tab of the Status field's properties in the ARS Admin Tool.

Only those fields that you code in your assignments are used to update or create the target form's record. When you create new records, you'll need to ensure that all required fields are assigned values.

Of course, Remedy server workflow fires. On submits especially, workflow may cause additional fields to be updated. In addition, workflow may reject the update with an error message.

Assignment References

In addition to constants, Meta-Update can use “references” in assignments.

These are references to data loaded from possibly different ARS schemas, from fields in an ASCII file, from LookUps, from parameters passed on the command line, from environment variables, or from references assigned during the Meta-Update session..

ARS and file records are tagged with a name. For example, the requester of a ticket may be tagged **`PplReq`**. Then fields from these forms can be referenced in assignments to the target form. For example:

```
Requester email     =     PplReq,      Email address
Requester name      =     PplReq,      Surname
Requester name      =     ", "
Requester name      =     PplReq,      GivenName
```

The above example shows concatenation of a text field’s assignment.

A reference for an ARS record is the Tag that identifies the record (and defines the Schema of that record) and either a field id or a field’s ARS database name.

String References

A string reference is used in queries and other control statements. It comprises a mixture of text and assignment references wrapped in dollar signs.

```
'Requester email' = "$PplReq, Email address$"
'Requester Name' LIKE "%$File, Surname$, $File, Firstname$%"
'Requester Name' LIKE "%$File, Surname$$File, Firstname$%"
'Requester Name' LIKE "%$File, Surname%"

Server           = $ ENV, ArsServer $
```

Loads

You tell Meta Update what data records to load and what names you want to use for these records.

```
LoadQ    =   PplReq,     CTM:People,   "'1' = "000000000041306"
```

Loads are processed based on a Query. In this case, the query must result in one and only one record.

```
LoadQ    =   PplReq,     SHR:People,   'People ID' = "$Arg, PplId$"
```

Loads are processed in the order that they are coded and all the loads are completed before the assignment process begins.

Note that Loads may be replaced by LookUp assignments. These allow multiple records, default values in case of no matches, and cache records in memory.

Types of Sections

This list summarises the types of sections used by Meta-Update.

Main	Gives the update ARS server and sign on information and the Meta-Update licensing information.
Read Servers	Specifies additional read ARS servers and sign on information.
Control	Specifies the operations you want Meta-Update to perform. These are the heart of Meta-Update scripting.
File	Defines the format of an external ASCII file.
Field	Specifies the fields and field interpretation / transformation rules of an external ASCII file, an SQL row, a regular expression extract.
Assignment	Contains the actual field assignments to be made to the target form. Also used to assign script variables and control script execution.
LookUp	Used in an `@LookUp` assignment to provide a mechanism for translating data values.

Control Sections

When you fire Meta-Update, you tell it which section is the first section to read in the script file. This is like a “Main” in a program.

A Meta-Update control section and gives information about the operation you want Meta-Update to perform, ultimately causing records to be added, updated, or merged, in an ARS server.

Iteration

A section can perform its function exactly once or can iterate through a set of records or values applying its output each iteration. Iteration may be based on:

- An ARS query
- A Direct SQL Query through the ARS API
- A columnar, ASCII file
- A list of values in a string
- A diary field value
- The set of fields in a Tag or record
- A while loop

In the cases of Files and SQL you can set up interpretation and translations of the data. For example, an ARS time field from a direct SQL column or a file column can be interpreted and assigned to time and character fields resulting in a proper time stamp value.

An Iteration may be terminated prematurely when an **`Until=`** condition is true.

Output

The control section specifies the ARS output operation Meta-Update will perform: Each section’s iteration will cause exactly zero or one output: an update, create, or merge of an ARS record, or file. A sections output:

- May do nothing, so that it’s assignment sections can fire,
- May always create a new ARS record,
- May create a record or update an ARS record based on a query, creating one if the update query returns no records.
- May output a new file of a multi-pattern file, or a new record in a single pattern file, or a new record in a columnar file.

Assignment Sections

Control sections specify sets of assignment sections that are called at different times during the control sections iteration or to handle the output assignments. Assignment sections are listed with different keywords and a control section. They:

- specify the assignment sectionss to be applied to the target update record for create or update, or for the file output
- fire once when the section starts
- fire after the next iteration record or value is loaded but before an output is applied
- fire if an update is applied successfully
- fire if an update had an error
- fire after an update is applied but before other sections are launched
- fire after all Launches are processed
- fire once on section termination.

Assignment sections can load records, SQL data, files, and transform data. They can launch client or server processes. They offer nested ifs, includes, regular expressions, arithmetic, date operators. In short, they offer a rich facility for transforming data.

Launching or Nesting

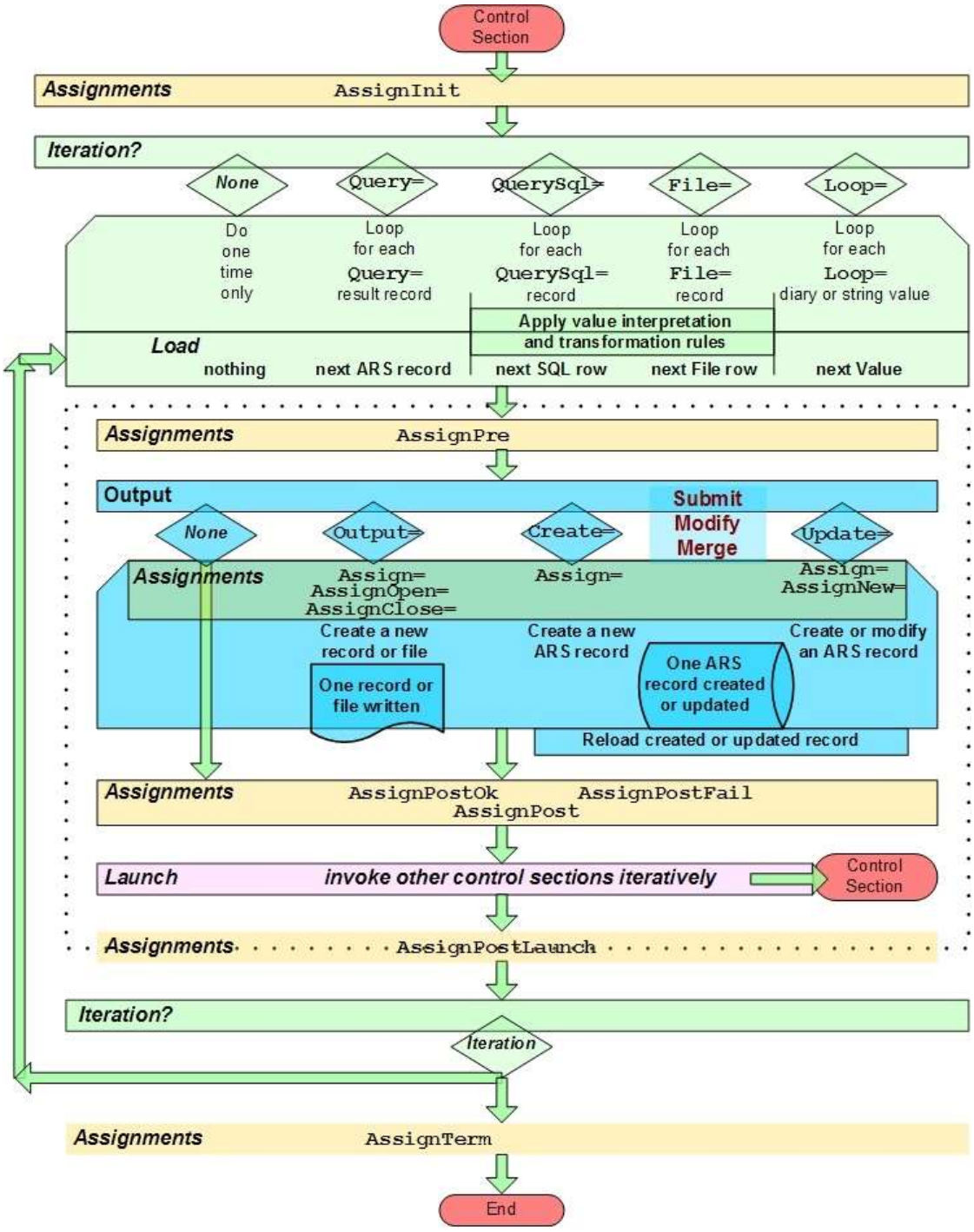
Finally, a section can launch other sections. All previously loaded references are available to the inner, launched section. These launches can be conditional.

The Launch is the key to the power of Meta-Update scripting. When you launch a section, all of the references in all the previous sections is available to the launched section.

The update record is reread after the update to ensure all workflow results are available to the launched section.

Control Section Flowchart

This flowchart summarises a Meta-Update control section's operation.



Iteration

A Control Section may either iterate or not. If it does not iterate, it performs its cycle once and once only. If it does iterate, it loops, picking up a new record during each iteration, and performing the actions specified, which can include creating or updating other records, and launching other control sections.

The absence of an iteration statement, tells Meta-Update that the control section wants to execute once and once only.

This control section may itself be launched from a control section that iterates. In that case, it will perform operations once each time it is launched.

A control section can iterate on at most one of the following statements:

Query Allows you to iterate through the results of an ARS query, specified as per the Advanced Query Bar in the BMC User Tool. Of course, the full power of Meta-Update references is available for your query.

All records returned by the Query will be processed no matter what the server limit is set to. If the Query returns more records than the server limit, the Query itself will be chunked. The records are retrieved in blocks of 100.

One can control the starting record number and the maximum number of records of the Query results to process.

QuerySql Similarly, you can issue a direct SQL query through the ARS API, iterating through the resulting rows, and, naming, interpreting, transforming the resulting columns.

Remedy imposes no limit on SQL queries and returns all results when the query is returned.

File Any type of columnar file can be iterated through. Values can be transformed or interpreted.

CSV's that Excel or the BMC Import Tool can't handle, are not a problem. For example, CSV's with values having embedded new lines, stray, undoubled quotes, or different delimiters, are all supported.

Loop Loops may be based on:

- the entries of a diary field, allowing you to create records for each diary entry for example
- delimited strings, allowing you to process a Group List, or other encoded value, such as lists of tables
- any arbitrary condition being true
- the set of fields in any Tag or Remedy record
- the set of forms that make up a Join form

A control section's iteration can be stopped before the next iteration begins with an **`Until`** statement. If an Until statement is coded, the iteration continues while the until condition evaluates to true.

Query

You can tell Meta Update to perform a query. Meta-Update will then iterate through the results of the query, loading the records one at a time, and create or update the same number of target records. If your query returns six results, you can update or create six records in the target form.

A query is similar to a Load. You tag or name the loaded record so that you can use its data in assignments, you name the query's source form, and you specify a query string.

A query may refer to the target server (as specified in the Main section) or to any read servers.

```
Query    =   PrdCode,    ProductCode,                    &
             'Product' = "Pkg1" AND 'Status' = "Active"
```

You can also use references in your queries.

```
Query    =   PrdCode,    ProductCode,                    &
             'Product' = "$\$001$" AND 'Status' = "Active"
```

Or:

```
Query    =   PrdCode,    ProductCode,                    &
             'Product' = "$SrcMasterProductSale, ProductCode$"
             AND 'Status' = "Active"
```

Any query acceptable to the Remedy User Tool can be used. With Remedy, you must specify field labels or field ids and *not* field names when describing fields within single apostrophes.

With Meta-Update you can use field labels, ids, or names in ARS Queries. When using Meta-Update references, only field names or ids are used.

The query is always done after all the load statements that precede it in the control section. Any loads following the Query statement are done after the record from the query statement is loaded but before the Update record is determined.

Other loads are possible during the processing of the assignments.

The number of query records returned are loaded one by one and can be referenced by the tag, "PrdCode" in the above example. As many new target records are created as there are results returned by the query.

Loads processed during the assignments are done during the results iteration, after the current query result record is loaded and associated with the query tag. These loads and the assignments may reference the query tag.

The placement of the `Query=` or `File=` specification within the control section is significant.

Only loads specified before the `File=` or `Query=` can be referenced in the File= or `Query=` statements. Only loads coded after the `File=` or `Query=`= can reference items from the `File=` and `Query=` records.

Note that only one of `Query=, QuerySql=` or `File=` may be used in any single control section.

QuerySql

You can also tell Meta Update to perform a query using direct SQL. Meta-Update passes the query you code to the Remedy Server. Meta-Update will then iterate through the results of the query, loading the records one at a time, and create or update the same number of target records. If your query returns six results, you will update or create six records in the target form.

SQL Columns can be named and interpreted. For example, ARS Date fields can act as integers (un-interpreted) or as dates.

```
QuerySql =  BseGuid,    @na,                                    &
            select instanceid                                   &
            from bmc.core_bmc_baseelement                       &
            where modified_date > $Vars, ArgDateEpoch$”
```

Or:

```
QuerySql =  TskCnt,     @na,                                    &
            Select count(*) from ctm_task                       &
            where RootRequestInstanceID = ‘$IncSrc, InstanceID$’
```

File

A `File=` can be used instead of a query. Meta-Update will iterate through the records of the file, loading the records one at a time, and create or update the same number of target records. If your file has six records, you will attempt to update or create six records in the target form.

A file is an ASCII file with columns separated by delimiters or in fixed positions. An example could be a .csv file created by Excel, or an Exchange Post Office dump.

Using a `File=` looks like this:

```
File     =   ExchSrc,    ExchangeFileDef,     File Name

[ExchangeFileDef]
File     =   d:\dta\ArsTemp\exchange.dat
Type     =   Delimited, “\t”
Fields   =   1
```

In this example, each record of the file is loaded and tagged with ExchSrc. The column names are taken from the first record in the file.

The data in the file can then be used as references.

The file name can be a reference or a constant. It must evaluate to a file name valid for the OS. The OS user running Meta-Update must have read access to the specified file.

Loop

A `Loop=` can be used instead of a query or file. Meta-Update will iterate through the values specified, loading those values one at a time into a reference tag you specify, and create or update the same number of target records. If your loop value is parsed into six separate values, you will attempt to update or create six records in the target form.

Loops can be performed

- on string values that are separated by a delimiter
- while a condition is true
- on Diary field values
- on the set of fields and values in a Tag
- on the set of forms that make up a Join form

An example of a string value separated by a delimiter could be the User Group List of the user form. You could, for example, create a record for each group in a user's group list.

A `Loop=` looks like this

```
LoadQ    =   SrcTT,    HPD:HelpDesk,   '1' = "$Arg, ID$"
Loop     =   Diary,    sTag,           $SrcTT, Notes$

LoadQ    =   Usr,      User,           '1' = "$Arg, ID$"
Loop     =   String,   sTag,   ";",    $Usr, Group List$

Loop     =   While,    (exp)

Loop     =   Fields,   sTag,           SrcTag

Loop     =   Join,     sTag,           BMC.CORE:BMC_Mainframe
```

In the first example, a Help Desk ticket is loaded into the Tag `SrcTT`. The Notes field, a diary field, is parsed, and each entry in that diary field is iterated through. When the entry is loaded, the following references are made available to the section:

sTag	**User**	the login name of the user who made the diary entry
sTag	**Date**	the date of the entry: yyyy/mm/dd hh:mm:ss
sTag	**DateMdy**	the date of the entry: mm/dd/yyyy hh:mm:ss
sTag	**DateDmy**	the date of the entry: dd/mm/yyyy hh:mm:ss
sTag	**Text**	the entry text

The Date value is useful for assignments. This is the format that Meta-Update expects for date variables. The DateXxx values are useful for ARS Queries which require that the date be formatted according to the machine's locale. In Windows, this is set at a machine level. On Unix, the local may be controlled by environment variables. The "C" locale, a default, is refernced by DateMdy.

In the second example, a User record is loaded into the tag Usr. The Group List field is parsed (based on the semi-colon seperator specified) into a set of single groups. Each of those groups is interated through. When each group is loaded, the following references are made available :

sTag	**Text**	the single group id as a string

A sort may be specified if desired otherwise the entries will be processed in whatever order they are specified in. In the case of a Diary field, that order is from least recent to most recent. In the case of a string, it is simply the natural order of the contents of that string.

In the third example, the expression is evaluated and if true the the sections assigments and outputs are executed. In this case, no tag is specified and no values are loaded.

In the fourth example, the Loop is executed for every field defined by the **SrcTag**. An **@info** reference command is assigned to the **sTag**.

In the final example, the Loop is executed for each form defined by the specified Join form. An **@info** reference command is assigned to **sTag**.

Until

Any iteration statement may be controlled with an **Until** statement.

An **Until** statement specifies a condition that, if true, causes the section to stop processing the iteration.

The iteration is stopped without an error as though the end of the iteration was reached.

If you need to stop with an error, issue an **Abort** command in an assignment section such as the **AssignTerm**.

So, for example, a While loop can be an Until loop:

```
[Loop]
Loop        =     @if(1)
Until       =     ("$V, quit$")
```

Output

Create

You must tell Meta Update whether you want to update an existing record, or create a new record.

If you are ***always creating*** a new record, in every iteration of a section, you code a `Create=`.

The `Create=` specified the schema for the record to be created in and the Tag that the re-read record will be referenced as. Because the record is reread after creation, the Request ID field will be available.

The assignment section is applied to a new record in the `Create=` form.

If you want to create a new record or update an extant record depending on a query, you can use the **`Update=`** statement.

Update

You may code a query to determine the update record. This query is coded in the `Update=` statement.

Assignment sections can be coded for both updating and creating a new record when the Update query returned no records.

When you do use `Query=` the record you want to update can be the result of that query, or, a new record loaded from a query that returns one and only one record.

The simplest case is: the result of the query *is* the update target.

```
Query    =   Tgt,   HPD:HelpDesk,  'Master Ticket Id' = "$$001$"
Update   =   Tgt
```

Here, an update will occur for each ticket satisfying the query condition.

In other cases, you'd need to load the update target record by issuing a different query based on data in the results of the `Query=`. This second query is meant to find the target record ID. It must return exactly one or zero records.

```
Load     =   Src,   SalesProduct,  "$$001$"
Query    =   Cat,   ProductCat,    'Product' = "$Arg, Product$"
Update   =   Tgt,   SalesItem,     'SaleId' = "$Src, SaleId$" AND &
                                   'Item'   = "$Cat, Item$"
```

Here, an update will occur for each SalesItem satisfying the query condition on the ProductCat form. Running the `Update=` query during the iteration of the results from the ProductCat form will retrieve the actual SalesItem ID to be updated.

When you use a `File=` the record you want to update must be loaded from a query.

You'll need to load the update target record by issuing a query based on data in current record of the `File=`. This query is meant to find the target record ID. It must return one and

only one record.

```
File        =   ExSrc,   ExchangeFileDef
Update      =   Tgt,     SHR:People,   'Email Address' = "$ExSrc, email$"
Assign      =   UpdPeople
AssignNew   =   AsgNewPeople

[ExchangeFileDef]
File        =           d:\dta\ArsTemp\exchange.dat
Type        =           Delimited, "\t", FldHdr
```

Here, an MS Exchange Post Office extract will be processed. Each SHR:People record with the same email address as in the extract - with changes in the data basing assigned - will be updated. If there is no SHR:People record with the email address, one will be created.

The `FldHdr` in the file definition section's `Type=` indicates that the first file record contains the field names. This is typical in Excel spreadsheets and in Exchange server extracts.

Output

A Meta-Update control section can also be used to create output ASCII files, either columnar, like a CSV for example, or text files like emails, XML, or, HTML pages.

A single Output statement may

- create a single file always appended to
- create a set of files appended to at different times
- create a new file on each iteration of the section

Fields can be defined for columnar files as well as field transformations. Fields can be copied from Schema.

This is useful for generating complex reports where data is gathered from multiple forms and records.

Launch

Meta Update allows you to follow chains of linked records. One control section can launch other control sections, which can, in turn, launch still others.

For example, let's say you have the following tables

Organisation *1:many* Sites *1:many* Services

You want to write a script to invalidate all Services belonging to an Organisation.

You write a Meta-Update control section that queries for the single Organisation record you wish to invalidate services for. This control section launches a second control section that queries for all sites associated with this Organisation.

That second control section processes a set of Site records and for each of those Site records, launches a third control section that queries for all services associated with that single Site record being processed.

That third control section invalidates the Services records for each Site of the Organisation.

Here is a simple file that will do this:

```
[Org]
Query   =  Org, Organisation, '1' = $001
Launch  =  Site

[Site]
Query   =  Site, Site, 'Organisation ID' = "$Org, 1$"
Launch  =  Services

[Services]
Query   =  Service, Services, 'Site ID' = "$Site, 1$"
Assign  =  ServiceInvalidate


[ServiceInvalidate]
Status  =  Inactive
```

Launches can be conditional. That is, a separate section can be launched if a condition is satisfied. That condition may be dependent on any of the data held in memory at the time the launch condition is evaluated. This includes data from any previous control sections, the current record from a query or file and the updated record. Further, the section name to be launched can be derived.

Multiple launches can be coded for any section and a launched section can in turn launch more sections.

Control Section Example

Two rather contrived examples may help to illustrate.

Example 1
In this simple example, we want a command line script that will raise a new ticket. We'd like the script to take one argument, a key to a configuration table that would give the details of the ticket to be raised.

Example 2
In this update, we process a Query of an "Update" with five columns: Schema, Key value, Key field label, field name, and field value. We want a script that processes that file, updating only that field in the right schema record and ensuring no workflow fires or modification dates are touched..

Example: Migrate any Table

This example will migrate any data table from a source server to a target server. The table name is an argument to the script.

The configuration specified things like ticket CTI, Priority, Summary and Description and so on. The script was passed the key value (a word) to look up in the configuration table. It would do the look up, and if specified, build a ticket as configured.

```
SthMupd.exe  Eg1-TT-Create  Req -p    Tkt-Cfg-Typ-1"
```

```
[Main]
ReadServers   =   ReadServer
Server        =   ArsDev
User          =   Demo

Arg           =   FormName

[ReadServer]
Tag           =   SrcServer
Server        =   ArsSourceServer.com
User          =   Demo

[Do]
Query         =   @SrcServer,
                  SrcRecord,
                  $Arg, FormName$,
                  1=1
Update        =   Upd,
                  $Arg, FormName,
                  '1' = "$SrcRecord, 1$"
Merge         =   Yes, No Workflow
Assign        =   Do-asg
AssignNew     =   Do-asg

[asg]
@Cmd          =   Copy, SrcRecord,
                  CoreAssign
```

The **[Main]** section identifies the ARS server and sign on parameters. It also gives the script argument names and usage information.

The **[ReadServer]** section identifies the "Source" ARS server and sign on parameters.

The **[Do]** section is passed on the Meta-Update command. It queries the source server for all record in the passed ARS Form Name and migrates the data – as is - using the Merge API.

The **[Do-asg]** section holds the individual field assignments for the updated record. In this case, we will copy all the data fields from the source record into the target record.

Installing

Installing Meta-Update

Expanding the Distribution

Meta-Update is distributed as a single zip or g-zip file.

The file may be expanded in your applications area. For example,

On Windows, you could unzip the distribution in:

```
"C:\Program Files\STH\"
        Or
"C:\Program Files\SoftwareToolHouse\"
```

The zip creates a single directory containing all Meta-Update software, documentation, and samples as described below.

The root directory is called Mupd_Xxx where X.xx is the Meta-Update release. This allows you to test different versions and APIs of Meta-Update easily.

Complete Installation

There is no formal installation for Meta-Update. Meta-Update is meant to be run from a command line or shell or fired by ARS workflow, or other automated processes.

By expanding the distribution file, you have completed the installation of Meta-Update.

You may choose to include the Meta-Update distribution directory on you path and your library path. See Runtime Environment below for more information.

Distribution Contents

The Meta-Update distribution has a single directory at the root.

The single directory is called **Mupd_X_xx** where **Mupd_X_xx** is the Meta-Update release. This allows you to test different versions and APIs of Meta-Update easily.

This directory contains directories and files that is the Meta-Update distribution.

Directory	Contents
docs	This directory contains the Meta-Update user manual and the Trace Facility Administration Guide. These are distributed as two PDF files.
samples	Contains sample Meta-Update scripts and a sample Trace configuration file.
bin	Contains all required binaries (.exe) and libraries (.dll) for Meta-Update and bundled utilities. This is the 32 bit path. It is the only path that can be used on computers that have a 32 bit architecture. For computers with 64 bit architectures, Software Tool House recommends that the 64 bit binaries be used. See below for the bin64 directory. It is recommended that this path be added to the system Environment Variables for use by a Server, and in the User's Environment Variables for workstations.
bin64	This contains the x64 bit versions of binaries (.exe) and libraries (.dll) for Meta-Update and bundled utilities. This can only be used on x64 architectures. It is recommended that this path be added to the system Environment Variables for use by a Server, and in the User's Environment Variables for workstations.

The Meta-Update distribution includes a "**bin**" and "**bin64**" directory.

```
"C:\Program Files\SoftwareToolHouse\Mupd_5_56\bin"
"C:\Program Files\SoftwareToolHouse\Mupd_5_56\bin64"
```

This "**bin**" and "**bin64**" directories contain all required binaries (.exe) and libraries (.dll) for Meta-Update and associated utilities.

You *must* use the "**bin**" directory on 32 bit computers and we recommend you use the "**bin64**" directory on 64 bit computers.

It is recommended that this path be added to the system Environment Variables for use by a Server, and in the User's Environment Variables for workstations.

All Meta-Update binaries print usage instructions when entered with no arguments.

The Meta-Update distribution contains the following binaries for both 32 bit and 64 bit architectures:

File	Contents
SthMupd.exe	Meta-Update. - using local trace. This single executable is Meta-Update. It appends to a single log file – **SthMupd.log** – on each run.
SthMupdTrc.exe	Meta-Update. - using global trace. This single executable is also Meta-Update with a communication based for logging facility. It will either behave as **SthMupd.exe**, or, if the **SthTrcDaem.exe** process is running (see below), will send its logs and messages to that process. **SthTrcDaem.exe** can be configured and controlled. Note that Meta-Query, Meta-Delete, Meta-Schema are also offered in this trace facility by suffixing "Trc" to the binary names.
SthMqry.exe	Meta-Query This tool allows you to issue ARS and SQL queries through the ARS API and print or create CSVs from the results. Please See the Meta-Query User's Guide for more information.
SthMsch.exe	Meta-Schema This tool allows you to find information about the ARS forms and fields on your server and print or create CSVs from the results. Please See the Meta-Schema User's Guide for more information.
SthMdel.exe	Meta-Delete This tool allows you to delete records from ARS tables on your server. Please See the Meta-Schema User's Guide for more information.
SthLicUpd.exe	License Updater and Password Encryptor This utility is used to generate an **SthLic.cmd** or **SthLic.sh** file that sets environment variables for ARS server, authentication and Meta-Update licensing. It is also used to encrypt ARS authentication passwords.
SthTrcDaem.exe	The Trace facility daemon. This binary can be started when the machine powers on and left to run until the powers off. See the Software Tool House, Trace Facility Administration Guide for more information.

SthTrcCtl.exe	The Trace daemon control program. This program is used to control the trace levels, files, sizes, and so on. It communicates with the **SthTrcDaem.exe** process.
SthTrcEcho.exe	The Tracy utility that can be used to write messages to the trace facility.
SthArsTime.exe	Allows easy conversion of Remedy time stamp values.
SthSiniGet.exe	Allows extracts of script files to be used in batch files and performs a simple validity test on scripts.

Running Meta-Update

Running Meta-Update

In this section, we will cover:

- Setting up the run time environment
- BMC Remedy API versions
- Meta-Update program versions
- Using the license keys
- Environment variables
- The Meta-Update command line usage
- Meta-Update output and return values
- Meta-Update Tracing

Software Tool House Inc

Run Time Environment

Meta-Update runs in a Windows "Command Prompt" or UNIX shell.

Scripts and files developed and referenced may be interchanged freely between Window and UNIX.

Meta-Update scripts can be run

- manually in a shell or command prompt
- in a filter with the $PROCESS$ actions
- through a batch file or shell or Perl script
- through an OS scheduler like **cron** or **at**.

The runtime environment is the same for workflow, script, and manual operation.

The Meta-Update "**bin**" and "**bin64**" directories contain all required Meta-Update binaries or execuable programs, shared objects and dlls. The Meta-Update bin directory should be on the path.

On Windows, one of the two the Meta-Update "**bin**" and "**bin64**" directories needs to be in the PATH= environment variable.

```
Set PATH=D:\Apps\Sth\Meta-Update-5.56\;%PATH%
```

The program operates in a Command Prompt, or "DOS Box", or as a fired process. Local trace files are written in the current working directory by default.

On Solaris or Linux, one of the two the Meta-Update "**bin**" and "**bin64**" directories needs to be in the PATH= and LD_LIBRARY_PATH= environment variables.

```
 export PATH=D:\Apps\Sth\Meta-Update-5.56\:$PATH
 export LD_LIBRARY_PATH=D:\Apps\Sth\Meta-Update-
5.56\:$LD_LIBRARY_PATH
```

The program operates under any of the available shells or as a spawned or background process. Local trace files are written in the current working directory when not specified.

BMC Remedy API Versions

Meta-Update is generally compiled against the most current BMC supplied version of the BMC Remedy API. The Meta-Update distribution includes all BMC supplied dlls that are required.

The Meta-Update API version does not need to match the version of the servers that Meta-Update establishes with. Meta-Update can establish multiple connections to different Remedy servers of different releases.

Software Tool House always recommends that the highest API version is used no matter what your server version is.

Program Versions

There are two versions of Meta-Update and bundled utilities with different names. One is used for local tracing and the other includes tracing through a trace server. These programs have different names. They are the same name in all operating systems:

SthMupd.exe	Local trace version
SthMupdTrc.exe	Trace server version

Logging is controlled by the Meta-Update **-d** switch in the same way across versions. See The Command Line below for more information on the **-d** switch.

The local trace version always appends to a file named **SthMupd.log** in the current directory unless the trace file is named with the **-d** switch.

With the Trace server version, traces are sent to the trace server. The trace server is administered to record selected levels of traces and discard other levels. The trace server version, needs both the **-d** switch, and the trace daemon set correctly for debugging traces to be captured

The trace server must be running on the same machine as Meta-Update. Communication to the trace server is with the standard message queue facility under Unix or with Named Pipes under Windows.

If the Trace Server version of Meta-Update is run, and the trace server is not started, Meta-Update will act as though the local trace version was run. That is: a file named **SthMupd.log** in the current directory is appended to unless the trace file is named with the **-d** switch.

More information can be found on the Trace facility in Server Tracing below, and the document, The Common Trace facility.

The License Key

You need a license key to run Meta-Update. Please see Licensing below for more information on licensing Meta-Update and obtaining License Keys.

You can tell Meta-Update the license key in one of three ways:

- Code it in the script itself.
- Set an environment variable with it as done with **`SthLic.cmd`** and in the samples.
- Set it in a special form and record on the target ARS server.

In the latter case, the **`STH:License`** form must have a valid licence key. This form needs to be on the Script's Target ARS server. In the first two cases, the overhead of an extraneous Query and Get from the Remedy ARS server will have been avoided.

The environment variable to be set is **`SthMupdLic`**. In the script, you can specify **`License=`** in the **`[Main]`** section.

A utility is used to generate an SthLic.cmd Windows batch file, or SthLic.sh bash shell script. This is a convenient way to set licensing, server and authentication parameters. It also allows ARS User passwords to be encrypted. See SthLicUpd Maintenance Utility below.

Environment Variables

Both Meta-Update and the BMC Remedy API can be affected by using Environment Variables[1]. This section defines the Meta-Update environment variables and the values and behaviours associated with them.

BMC Remedy documentation is the accurate source for documentation on the BMC API environment variables. We summarize them here because they affect Meta-Update behaviour.

Meta-Update environment variables are fully defined below:

Environment Variable	Description
`SthScriptPath`	A path-like environment variable for finding Meta-Update scripts and files.
`SthApiRetry`	Allows Meta-Update to retry API operations on any BMC Remedy API errors or during server outages.
`SthMupdLic`	Specifies the Meta-Update license key for the main server.

BMC Remedy API environment variables are specified in the BMC provided documentation. The usage of these variables may be changed at any time. This list is included for convenience and because it affects and overrides Meta-Update behaviours. Validate all usage of these variables with your Remedy documentation.

Environment Variable	Description
`ARAPILOGGING`	Generates two files in the current working directory of the running Meta-Update process. Conflicts will occur when multiple Meta-Update processes with this environment variable are run.
`ARTCPPORT`	Sets **all connections** TCP Port to the servers. Overrides the Meta-Update `Port=` keyword which can be different for different servers.
`ARRPC`	Specifies a private RPC port for all server connections.

Script Path Environment Variable

Scripts may be specified on the command line or may be found by searching an `SthScriptPath` environment variable.

SthScriptPath is set the same way as PATH according to the OS that Meta-Update is running on.

On Windows, one could set the script path like this:

```
set SthScriptPath=E:\Projects\ITSM\Scripts;D:\Apps\STH\samples\;
```

[1] "**Environment variables** are a set of dynamic named values that can affect the way running processes will behave on a computer." - Wikipedia

On LINUX, one could set the path like so:

```
export SthScriptPath=/Projects/ITSM/Scripts/:/Apps/STH/samples:
```

Note the difference in the path and directory separators.

Subdirectories in the paths are not searched. However if the script passed to the command line contains a relative path, that relative path will be checked against the **SthScriptPath** and the first matching file will be opened.

API Retry Environment Variable

A Meta-Update job normally returns any errors received from the ARS server during any of its API calls and cancels the single record it was processing. It would then continue with the next record.

It is useful to protect the Meta-Update run from a server timeout, crash, or restart. Meta-Update can retry some API calls to the server based on configurable ARERR codes, a maximum number of retries, and a delay between retries.

The environment variable SthApiRetry= may be used to specify these retry settings.

Without this environment variable, all API calls that fail cause an error in Meta-Update that can result in a record being lost, not found, or the Meta-Update job terminating before processing all records of a query.

The **SthApiRetry=** string is either a single or multiple sets of three numbers:

```
start_ARERR_number [ - stop_ARERR_number ] Retries Delay
```

start_ARERR_number [- stop_ARERR_number]	Single or ranges of ARERR numbers can be specified.
Retries	A **Retry** count of 0 means infinite number of retries.
Delay	The **Delay** is in seconds. A **Delay** of 0 means no delay.

The following example illustrates its use to protect against servers crashes and servers that have timed out.

```
set SthApiRetry=90-92 0 60 93 0 30
```

```
export SthApiRetry=90-92 0 60 93 0 30
```

These examples retry API calls resulting in error 90, 91, 92, 93, retrying an infinite number of times, with a 30 second delay on ARERR 93 (timeout due to busy server) and a 60 second delay for ARERR 90, 91, 92.

Note that for Query timeouts (94), retries will generally not resolve the problem. Instead use the **TimeOutLong=** keyword of the **[Main]** section.

fs

License Environment variable

```
SthMupdLic    =      license-key
```

If this environment variable is defined, the license check is made against the value associated and no read is performed on the ARS server.

This is primarily used on the server and also in high performance situations.

```
      AnyVar        =      Value
```

Any environment variable may be used in a Meta-Update script. All defined environment variables are referenced by the reserved tag, **`ENV`**. The field name is the environment variable name.

Environment variables, like all other field names are case sensitive.

```
      Loop          =      String, Pth, ";", $ENV, PATH$
```

The above example loops for every directory in the PATH environment variable.

As another example, the environment variable, **`ArsGlobals = 5`**, could be used to load a site-specific set of values and keys to other records.

```
      LoadQ  =      Tag, Schema, '1' = $ENV, ArsGlobals$
```

The Command Line

A Meta-Update command at a minimum specifies the Meta-Update script and the starting section within that script.

That script may require arguments and Meta-Update accepts built-in switches – for example to run the debugger or increase logging detail.

To maintain compatibility there are two forms of Meta-Update command lines that can be given.

The older, deprecated, command line uses unnamed positional script arguments coded in the order of the script's **Arg=** statements and following a **–p** place marker.

```
>>> SthMupd.exe 090-SvrAdmin\220-SwLogs.ini Do –p tst1
I terminating successfully in 2 sec.
```

The new command line uses named arguments that can be coded in any order before or after the script and section.

```
>>> SthMupd.exe 090-SvrAdmin\220-SwLogs.ini Do –log tst1
I terminating successfully in 2 sec.
```

Either form of the command line may be used on any script. Software Tool House recommends the use of switch based arguments for clarity.

By convention, in this document and in our samples, script arguments are specified after the script file and section name.

```
>>> SthMupd.exe 090-SvrAdmin\221-SwLogs.ini Do
Line 28 - required argument -log not on command line; no default
specified
E . Function:
E . This is a Meta-Update script that switches the ARserver log files
E .
E . Usage
E .   SthMupd   221-SwLogs Do   -log  xxx
E .     where       xxx         is a log file name without a path
E .                               and without the .log
E .                             The path and ".log" are configurable
E .                               in the script
E . Examples
E .   SthMupd   221-SwLogs Do   -log  my
E .             will set all log files to:
"/apps/bmc/ARSystem/db/my.log"
E .
E terminating unsuccessfully in 2 sec.
```

Meta-Update has a set of switches that may be specified on the command line. Each script can also define a set of arguments that may be set on the command line.

Entering the Meta-Update command with no arguments yields usage help. Entering the Meta-Update command with the single –help switches yields more detailed help.

```
SthMupd.exe
SthMupd.exe  -help   |   more
```

Switches

Entering the Meta-Update command with no arguments or the single **-help** switch yields usage help.

```
SthMupd.exe
SthMupd.exe     |    more
```

Logging	
-d	Specifies logging. By itself, all specified full debugging logs to the default log file with no ARS Server logging and no Debug2 logging.
--d	As above but includes Debug2 logging and ignores any Trace assignment commands in the script.
-q	Inhibits echoing of specific logs to the console but does not affect the logging file.
-v	Verbose. Equivalent to –d:qas All field structures, queries, and data values are logged.
Development switches	
-e	Single error mode. Stops execution of the script when the first error is encountered.
-g	Debugging more. Enters the Meta-Update debugger.
Server switches	
Note that servers and authentication may be specified on the command line, in the script, or default to the environment variables set by the **SthLic.cmd** batch file. Defaults for the Main server when not coded on the command line or in the script are the environment variables:	
-server xxx	Specified the main ARS server. May be an IP or machine name. May also point to a specific server of a load-balanced server group.
-port xxx	Specified the main ARS server's port number. Zero is the default and indicates that the port mapper is used.
-user xxx	Specified the main ARS server's login user that Meta-Update will be running under. Note that this user is generally an administrator.
-password xxx	Specified the ARS user's password. May be plain text or encrypted with **SthLicUpd.cmd**.
Other switches	
-help	Summary usage instructions.

ArsSvrAdmin	The server name or IP.
ArsPort	The server port. Use of the port mapper is the default and can be specified with zero.
ArsUsr	The ARS user that Meta-Update will be running under. Note that this user generally has administrator rights.
ArsPwd	The encrypted or plain text password of the ARS user that Meta-Update will be running under.

Usage Help Text

```
Meta-Update    Version 5.56 (x64) for ARS lib 8.1.2
           (c) Copyright 1996-2015 by Software Tool House Inc.
               www.softwaretoolhouse.com

Function:
  SthMupd runs a Meta-Update script at the specified section
              against a BMC Remedy Server.

Function:
  SthMupd runs a Meta-Update script at the specified section
              against a BMC Remedy Server.
  See: http://www.softwaretoolhouse.com for the User's Guide and Licensing.

Synopsis:
  SthMupd [ switches ]    script-file    section    [ script-arguments ]

    The script-file and section must follow each other.

    Switches and arguments have the form:    -switch   [ value ]
    The script can include named arguments which are specified by using the script's
      argument name as the switch followed by the value for that argument. The script
      should explain its usage when run with Meta-Update with no switch arguments.

    script-file       is the Meta-Update script to run; may be found in the path-like
                        Environment Variable: SthScriptPath
    section           a section to process in the script file  ("Do" for samples)

    switches for logging;  Warning: Produces large output and slows throughput.
      -d           Full tracing into SthMupd.log with no '2' or ARS server tracing
     --d           Full tracing like -d, plus: '2' and ignores script Trace commands
     -d:x,y,f      Tracing:  x   specifies tracing levels: qsad2flp
                             y   ARS client tracing flags: fsap
                             f   is the tracing file name (local or Caution: global)
      -q,-quiet    Quiet:         inhibit all output to stdout (not log!)
      -v           Verbose:       same as -d:qsa
    switches for script development:
      -g           Debug Mode:    enter script debugger; "help" for commands.
      -e           single Error:  terminate job on first error (for script dev/test)

    switches for specifying servers         Note that servers must be licensed.
                                            Set defaults with SthLic.cmd
      -server    server    the [Main] server                default: ENV, ArsSvrAdmin
      -user      user      the [Main] server's ARS user       default: ENV, ArsUsr
      -password  Enc:xxx   the [Main] ARS User's password     default: ENV, ArsPwd
      -port      port      the [Main] server's ARS Port or 0  default: ENV, ArsPort

    other switches
      -help                      more detailed help (pipe to more)

Deprecated positional argument synopsis:
  SthMupd [ switches ]    script-file    section    [-p script-arg1 [ arg2 [ arg3 [
...] ] ] ]

For more details, execute:
  SthMupd -help

143704.690 i terminating successfully in 0 sec.
```

In the local trace version, the –d switch causes a high level of tracing. This data is appended to a file that will grow if not deleted occasionally. Without the –d, the file will still be continually added to, but at a much reduced volume. Only Error, and other informational messages will be written. See Tracing below for more information.

In the Trace Server version, the **-d** switch causes a lot of message traffic between Meta-Update and the Trace daemon. The trace files are cycled through and do not grow beyond the limits specified in the trace configuration. See ***Tracing*** for more information.

The **-q** switch indicates quiet operation. No messages will be echoed to the stdout or stderr files at all. This includes all Error and Info messages as well as the copyright notice. These messages will still appear in the logs.

The **-n** switch indicates a null operation. No database writes are performed but all queries and loads are processed. The assignments are also processed and the updating data is printed to the console. This may be very useful when you are developing a new script file. Note that with complex scripts, because no database writes are performed, references needed may not exist.

The **-e** switch indicates a "single error" operation. The first error that occurs will stop the run.

Normally, a file or query is processed and sections that are launched may succeed or fail. If a launched section fails, then the remaining records in the file or query continue to be processed. Using the **-e** switch changes that behaviour so that the job is ends when the first error happens.

When developing scripts, this allows the developer to sort out each section in sequence quickly.

The **script-file** parameter is the name of the file containing the Meta-Update controls and the target record assignments. It must exist and read access must be permitted for the user running Meta-Update.

The **ArSvr**, **ArUsr**, **ArPwd**, and, **ArPort** parameters will override similar parameters in the Main section of the script file. If they are not coded in the assignment file, they are required on the command line.

If **ArSvr** is coded, the **ArUsr**, **ArPwd**, are also required, and **ArPort** is required if the listed server does not use Port Mapper. The command line arguments cause the equivalent script file keywords to be overridden and ignored.

Generally, one would code these in the file and let the operating system's file security prevent unauthorised access to that file. This would keep the ARS User and password secure. In the script, these may be set to environment variables or other references.

The **-p** is simply a separator from the previous server, user parameters, and all the following parameters. These next parameters can be referenced in the script file. They can be referred to as $001, $002, and so on, or they can be named and referenced by those names. These passed arguments can be used as keys in loads, or as text in queries and assignments. Wrap long values in quotes according to your shell as needed.

Using Positional Arguments (Deprecated)

Following the script and section a **-p** switch is used as a script Argument place holder. This is followed by the argument values specified in the same order as the **Arg=** or **ArgNm=** statements in the script.

As this is deprecated, Software Tool House recommends updating any batch files or $PROCESS$ filters to use the new switch based command line syntax.

Program Return Values

The program returns a zero upon successful completion. If ***any*** errors occur, the program returns 1. This value may be used in scripts to decide a course of action.

Errors and important informational messages are reported the trace file. They are also echoed to stderr, generally the console.

stderr may be redirected. On UNIX and Windows, the syntax is the same:

```
SthMupd.exe    . . .      2>>errors.txt
```

Or

```
SthMupd.exe    . . .       2>errors.txt
```

The first command appends between runs. The second creates a new file each time.

This file may be examined with any ASCII editor such as Notepad, Word, vi… The format of the trace messages are explained further in Tracing below.

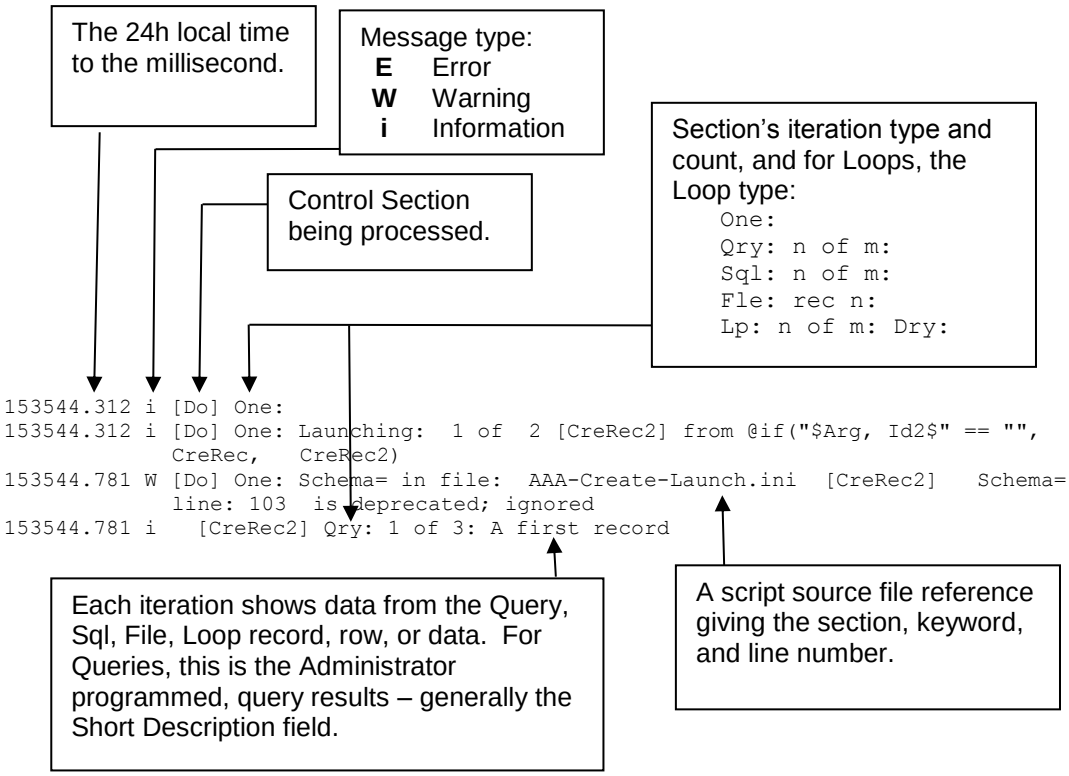
Note that error messages are also always written to stderr, which is generally the console window. If redirected as in the above example command invocations, Errors and Warnings may be grep'd or find'd from this file. See Tracing below for more information.

Program Output

Unless the –q switch is used, Informational, Warning, and Error messages are echoed to the console. These messages tell you what section is working on what record and lists outputs to ARS tables. These messages are also captured in the trace logs.

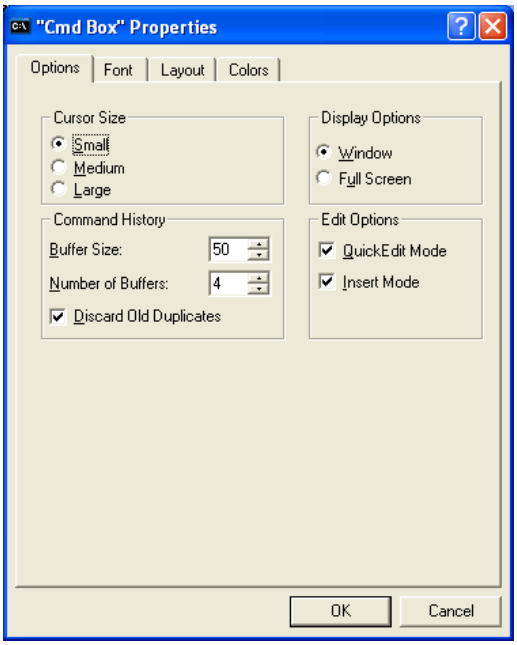
An example:

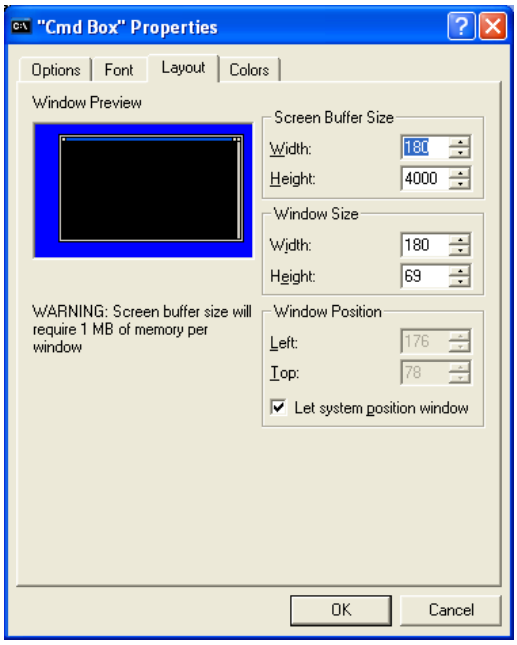
```
E:\Dta\_wrk\ > SthMupd.exe AAA-Create-Launch.ini Do -p 426 429

Meta-Update    Version 5.56 (x64) for ARS lib 8.1.2
           (c) Copyright 1996-2015 by Software Tool House Inc.
               www.softwaretoolhouse.com
153544.312 i [Do] One:
153544.312 i [Do] One: Launching:  1 of  2 [CreRec2] from @if("$Arg, Id2$" == "",
             CreRec,   CreRec2)
153544.781 W [Do] One: Schema= in file:  AAA-Create-Launch.ini  [CreRec2]   Schema=
             line: 103  is deprecated; ignored
153544.781 i   [CreRec2] Qry: 1 of 3: A first record
153544.890 i   [CreRec2] Qry: 1 of 3: Merged schema: _Test, Id: 000000000004474 OldId=
153544.921 i   [CreRec2] Qry: 2 of 3: and now, only seconds lat
153544.968 i   [CreRec2] Qry: 2 of 3: Merged schema: _Test, Id: 000000000004475 OldId=
153544.968 i   [CreRec2] Qry: 3 of 3: A second entry made a few
153545.031 i   [CreRec2] Qry: 3 of 3: Merged schema: _Test, Id: 000000000004476 OldId=
153545.031 i   [CreRec2] Qry: eof 3 record OK; 0 records with errors; total: 3.
153545.031 i [Do] One: Launching:  2 of  2 [CopyRec2] from @if("$Arg, Id2$" == "",
             CopyRec,  CopyRec2)
153545.031 W [Do] One: Update0= in file:  AAA-Create-Launch.ini  [CopyRec2]   Update0=
             line: 98  is deprecated.  Use AssignNew=
153545.031 i   [CopyRec2] Qry: 1 of 3: A first record
153545.125 i   [CopyRec2] Qry: 1 of 3: Merged schema: _Test, Id: 000000000004477
             OldId=
153545.125 i   [CopyRec2] Qry: 2 of 3: and now, only seconds lat
153545.187 i   [CopyRec2] Qry: 2 of 3: Merged schema: _Test, Id: 000000000004478
             OldId=
153545.187 i   [CopyRec2] Qry: 3 of 3: A second entry made a few
153545.234 i   [CopyRec2] Qry: eof 3 record OK; 0 records with errors; total: 3.
153545.234 i [Do] One: 1 record OK; 0 records with errors; total: 1.
153545.234 i Statistics:
153545.234 i         Sections:                    3
153545.234 i         Maximum section depth:       2
153545.234 i         Assignment Sections:         6
153545.234 i         Singleton Sections:          1   errors:      0
153545.234 i         Queries:                     2
153545.234 i         Query records:               6   errors:      0
153545.234 i         Output Schemas:              0
153545.250 i         Output Schema records:       6   created
153545.250 i         Output Schema records:       0   updated  (with 0 skipped)
153545.250 i         Outputs OK:                  6
153545.250 i         Outputs Errors:              0
153545.250 i         Outputs Aborts:              0
153545.250 i         Input   Errors:              0
153545.250 i terminating successfully in 1 sec.

E:\Dta\_wrk\ >
```


The 24h local time to the millisecond.
Message type:
E Error
W Warning
i Information
Control Section being processed.
Section's iteration type and count, and for Loops, the Loop type:
One:
Qry: n of m:
Sql: n of m:
Fle: rec n:
Lp: n of m: Dry:
153544.312 i [Do] One:
153544.312 i [Do] One: Launching: 1 of 2 [CreRec2] from @if("$Arg, Id2$" == "",
CreRec, CreRec2)
153544.781 W [Do] One: Schema= in file: AAA-Create-Launch.ini [CreRec2] Schema=
line: 103 is deprecated; ignored
153544.781 i [CreRec2] Qry: 1 of 3: A first record
Each iteration shows data from the Query, Sql, File, Loop record, row, or data. For Queries, this is the Administrator programmed, query results – generally the Short Description field.
A script source file reference giving the section, keyword, and line number.

Ideal Command Prompt Properties

Software Tool House recommends that for the convenience of the Meta-Update script developer, the Command Prompt have a wider and deeper buffer and that Quick Edit mode be set. This applies to the UNIX shell as well.





On Windows, click the Command Prompt Icon on the Title Bar, select Properties and ensure that QuickEdit Mode is on and then increase your Buffer Size Width and Height.

In addition, we highly recommend that “Cygwin” be installed, and Meta-Update script developers become familiar with it. There are numerous utilities that are especially useful for handling large log files.

“Cygwin” provides open source UNIX-like utilities and shells for Windows. It is available at www.cygwin.com

Tracing

Tracing can be controlled through the use of the –d switch. When a –d is specified with no additional options, full Meta-Update tracing is turned on. With `-d` no ARS client tracing is turned on.

With full tracing a great deal of data is generated. Without `-d`, only a very few messages will be traced.

Tracing levels for both Meta-Update and ARS can be specified with the –d: switch options.

```
-d  :  [ fpd2as , ] [ fsap ] [ , file ]
```

The first set of letters specifies the Meta-Update tracing levels. A comma is used to separate the Meta-Update levels and the ARS levels. The second set of letters specifies the ARS client tracing level. A further comma separates these levels from a specific trace file name.

If a full tracing switch is specified, further switches may be specified as the next set of parameters.

For Meta-Update tracing, the levels are specified with a single case sensitive character as follows:

```
S   Severe        Severe error
E   Error         Error
W   Warn          Warning
A   All           Always   like info but never masked out
R   Run           Run   execution instance
  Script Processing     These are on by default but may be turned off.
i   Info          Informational   (on by default)
  Script Debugging      These are echoed when selected with the -d
Q   Qry           ArQuery, Sql;     all query strings
G   Get           ArGet             all ArRecGet ids
U   Put           ArPut             all ArRecPut ids etc
  Debugging settings    These are never echoed.
  Caution:These generate masses of logs and can affect performance.
F   Func          Function entry and exit
d   Dbg           Debugging         detailed debugging
2   Dbg2          Debugging lvl 2   more details yet
a   Data          Data              data values: records, fields
s   Struct        Structure         data Structures
l   List          Script listing and files are logged
```

For ARS tracing, the user id the Meta-Update signs on the update ARS server must be in the Group that the ARS administrator has specified client side logging for in the Server Information panels using the ARS Administrator tool.

The following options can be specified:

s	SQL logging
f	filter logging
a	API logging
p	Plug-in logging

Specifying any ARS tracing implies Meta-Update tracing of level 2.

In the next example, we want the filter traces from ARS and the Meta-Update data traces. This will show us what value each field had before the ARS submit, set, or merge call, as well as the filter logs produced by that call.

```
-d:a,f
```

In this example, we want complete tracing, including complete ARS tracing, and we want to direct it to a specific file:

```
-d:,sfap,d:\trc\my-script.log
```

There are two versions of Meta-Update: one uses local tracing and produces a trace file in the current working directory of where the program is run.

Local Tracing

The local trace file is called **`SthMupd.log`** unless a file name is specified on the **`–d`** switch. **`SthMupd.log`** can be found in the current working directory of the Command Prompt or shell where Meta-Update was run from.

This file is appended to with each execution of Meta-Update. **`SthMupd.log`** will continuously grow in size. It is recommended that you delete the file before the next execution of Meta-Update.

There is no locking mechanism for multiple instances of Meta-Update running simultaneously in the same directory. This can happen when ARS workflow fires a Meta-Update process on the server.

It is recommended that if Meta-Update will be used in workflow, or in multiple, concurrent instances on a single machine, that the Trace server version be used. The Trace server must be running.

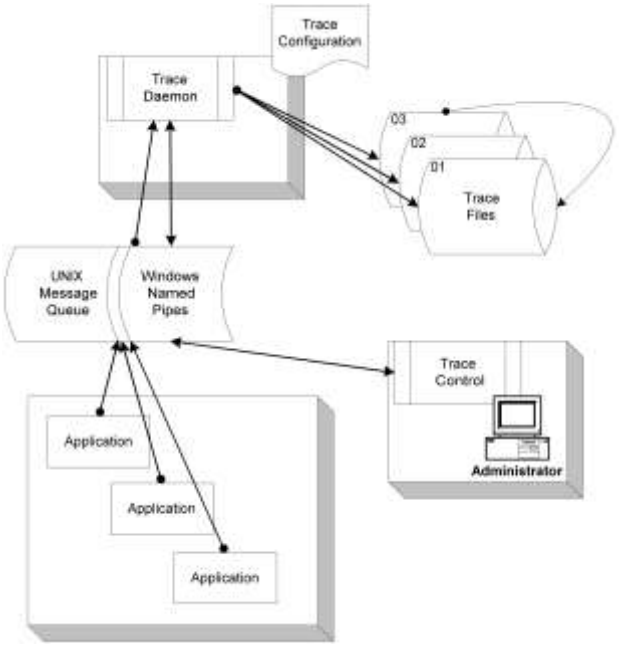
For ad-hoc runs of Meta-Update from a client machine it may be more convenient to use the local trace version.

When using the –d switch, a great deal of logging information may be written.

With or without full tracing, a file is created or appended to each Meta-Update is run. This file will grow in size. It is the user's responsibility to remove this file from time to time as appropriate.

Server Tracing

An alternative, communication based trace facility is available for high use applications. With this server based trace facility, the machine administrator manages the detail of the messages captured, and the size and number of trace files. Tracing is controlled independently of any application using it.



All client binary (executable) names that have the server based tracing included are suffixed with “Trc”. Meta-Update, for example, would be **`SthMupdTrc.exe`**.

If the trace daemon is not running, the same local trace file, **`SthMupd.log`**, is created or appended to. .

The following binaries are supplied with the server based tracing facility.

trcdaem.exe	This is the trace server itself. It should be started automatically when the machine starts.
trcctl.exe	This controls the trace daemon allowing the tracing levels to be set, switching to the next generation of trace file, and shutting down the trace server.
trcecho.exe	This utility adds records to the trace file and can be used in shell scripts or Windows command files.

Note that Meta-Update must be invoked with a **`-d`** switch for any debug level traces to be sent to the trace daemon. The trace daemon must also be set to capture the level of tracing desired.

The trace daemon uses a configuration file to specify both communication parameters and file handing and other trace daemon operational options.

All trace clients, such as Meta-Update or **SthMupdTrc.exe** for example, need to access this file to read the communication parameters. The location of this file is given by an environment variable.

On UNIX the trace daemon uses the POSIX message queue facility. The daemon should be run at a higher priority, or lower nice value, than any of its clients to prevent messages being lost. Further, system parameters should be adjusted so that the message queuing is not a performance bottleneck.

Under normal production usage (without the –d switch) very few messages are sent to the trace daemon and so performance is not generally an issue.

On Windows, Name Pipes are used to implement the inter-process communication. This will generally not require any system parameters to be changed to affect the performance. The trace daemon performance is not generally a bottleneck on Windows systems.

Note that to capture a level of trace messages beyond the minimum, both:

- ✓ The trace daemon is configured to include the desired trace level, or by using the trace control program. the desired trace level is on; and,
- ✓ The program will have been run with the **-d** switch specifying the desired trace level.

An environment variable is used by the trace daemon and all trace clients. This environment variable specifies a trace configuration file. The environment variable can be set in Windows as a system wide variable.

```
Set TrcIni=c:\etc\conf\SthTrc.cfg
```

The configuration file must exist. It is an ASCII file (created with Notepad or vi for example) and follows the format rules for a Meta-Update command file but with no section names. It can have these variables:

```
#  Trace facility configuration file for sth-m3
#    file:  e:\etc\conf\trace.ini
#
#    Environment variable must be defined system wide...
#      TrcIni=e:\etc\conf\trace.ini
#

QueueKey   =  e:\etc\conf\trace.ini
TraceFile  =  e:\trc\trace
GenMax     =  99
RecMax     =  500000
TrcLvl     =  dasfp2
TrcTme     =  30
ErrLog     =  e:\trc\error.log

ScOpen     =  cmd /c trcerrm.cmd
```

`QueueKey =` is used on Unix platforms only. The message queue is opened using the specified file's i-node as the key. On Windows this parameter is ignored.

`TraceFile =` specifies the fully qualified prefix for the trace files. The string specified is suffixed with .xx where xx is the current open trace file.

`GenMax`	=	specifies the maximum number of trace files to produce. Specifying 99, for example, would mean that a maximum of 100 files named e:\trc\trace.01, .02, .. .99 could exist at the same time. After trace.99 is filled up, trace.01 will become the current file.
`RecMax`	=	specifies the maximum number of records per file. When this number is reached, the trace file will be closed and the next trace file will be opened.
`TrcLvl`	=	the starting trace level. See trcctl.exe for more information about the levels and their meanings.
`TrcTme`	=	a normal trace client is presumed to live for a short time between issuing traces. Long lived processes may have larger amounts of time between traces. This specifies the maximum amount of time between calls for the trace daemon to consider that the client program has failed or been aborted without a proper shutdown. When this time is reached, an error trace message will be added to the trace file and client resources will be freed.
`ErrLog`	=	specifies a single file that will collect R and E messages. This file will always grow. It is the administrators responsibility to remove the file on occasion.
`ScOpen`	=	can be used to run a single command file or shell script. It is passed the version number of the file just closed and the fully qualified file name: In the above example, when the trace switches (say it was on 19 and is now on 20), a command will be run in the system as follows: `cmd /c trcerrm.cmd 19 e:\trc\trace.19`

See www.softwaretoolhouse.com for more details and for the Trace User document.

Trace Format

A trace record looks like this:

```
hhmmss.nnn f 0pid Prog     text
```

The hhmmss.nnn is the time that the record was created by the application. Note that trace records may appear out of sequence between applications but will never be out of sequence for any one instance of an application. Note also that a single application may have two instances running concurrently.

The f is the highest priority TrcLvl value on the Trc call that sent this trace message. Values are as follows:

```
S   Severe        Severe error
E   Error         Error
W   Warn          Warning
A   All           Always   like info but never masked out
R   Run           Run   execution instance
  Script Processing    These are on by default but may be turned off.
I   Info          Informational   (on by default)
  Script Debugging     These are echoed when selected with the -d
Q   Qry           ArQuery, Sql;     all query strings
G   Get           ArGet             all ArRecGet ids
U   Put           ArPut             all ArRecPut ids etc
  Debugging settings   These are never echoed.
  Caution:These generate masses of logs and can affect performance.
F   Func          Function entry and exit
d   Dbg           Debugging         detailed debugging
2   Dbg2          Debugging lvl 2   more details yet
a   Data          Data              data values: records, fields
s   Struct        Structure         data Structures
```

The 0pid is the process identifier, in hexadecimal, of the process that generated the trace message. This number can be used to select the trace records for a specific instance of a specific application.

The Prog is the program name coded on the application's TrcInit call. Each application that uses the trace facility should document its use of the facility in its User's Guide. You can use this field to extract those records written by any one application.

The text is the actual text of the trace message and is entirely application dependent.

Firing from Workflow

Meta-Update may be fired from workflow as Run Process or Set Fields $PROCESS$ filter or active link.

When firing from workflow on the server, the environment is that of the ARS server process. It is prudent to code a script or batch file in the workflow and then have that script or batch file set up the environment for the run, invoke Meta-Update, and possibly do some termination activities.

The environment generally includes a path to the executable and to any required shared libraries or dlls, other environment variables, parameters, and the working directory.

As workflow is fired at independent times, it is possible for multiple copies of Meta-Update to be running simultaneously. If so, the Server based tracing version is highly recommended to properly serialise log files.

Developing Scripts

Normal Meta-Update runs will report script errors with an 'E' level message echoed to the console. That message will print the script file name, section, line number, and, if appropriate, the keyword being processed.

```
114159.531 E [Do] [asg-init] AssignInit apply was aborted in file:
               FD-SupGrp-Ren.ini  [asg-init]   @Cmd=  line: 74
```

Errors may be caused by different things:

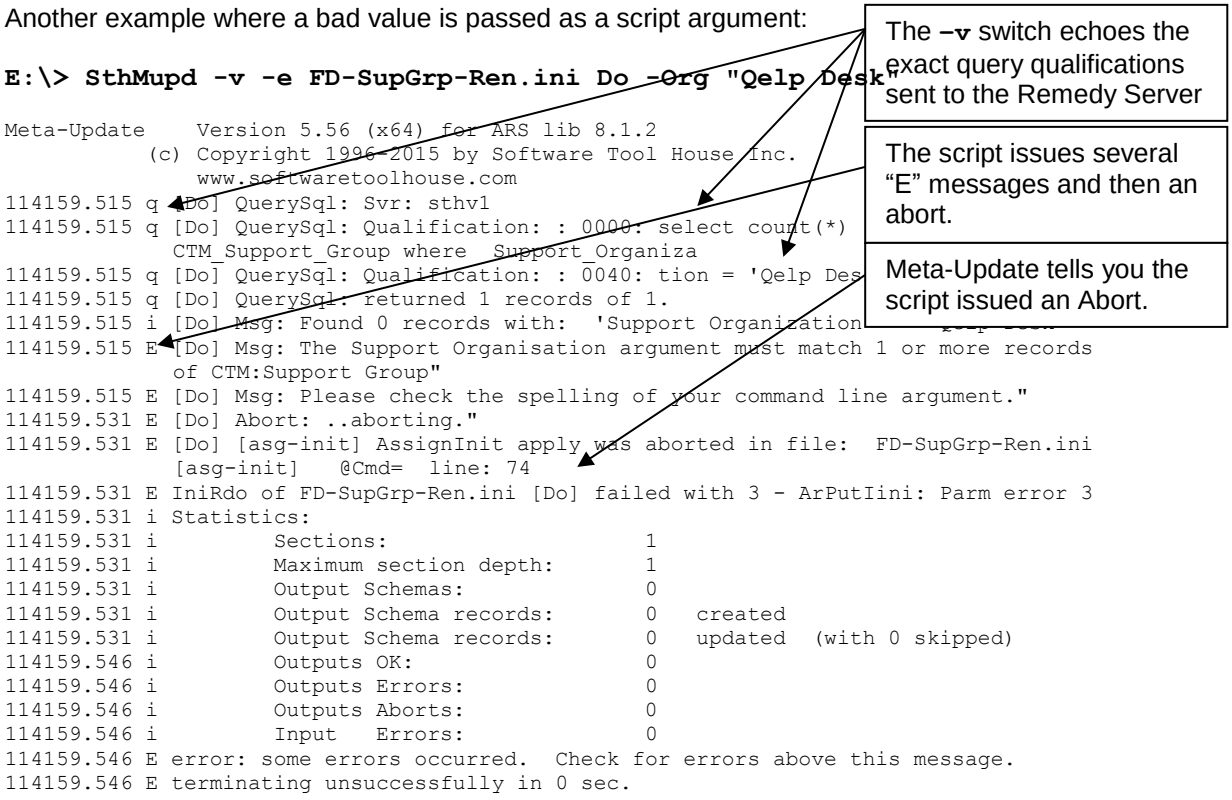
- Syntax errors
- ARS reported errors such as unrecognised schema names or field names or labels
- LookUp or Load failures
- User Aborts

Meta-Update has several switches that will aid in script development which would normally not be used in production runs.

-e	single Error	With this switch, any error in any section will stop the run. We recommend you use this switch when you develop and test scripts. You will generally not want it on production runs.
-v	Verbose	This prints all query qualifications and results to the console and to the log file. We recommend you use this switch when you develop and test scripts. You will generally not want it on production runs.
-n	null	This switch prevents any ARS updates or creates. This is only useful for the most simple of scripts as generally launched sections depend on access to a previous sections updated and reread record reference.
-d	Logging: Debug	This should not normally be needed. It is intended to be used when using Meta-Update support. It provides complete debug level information on the job and generates masses of logs. You can also specify you want ARS client logging with this switch. See ***Tracing*** above for more information.
-g	Script Debugger	This invokes the Meta-Update script debugger. The script debugger allows you to set breakpoints and single step though your script's operation. You can get debugging help, print your script, examine references, control breakpoints, and resume normal execution. See **Script Debugging** below for more information about using the Meta-Update Script Debugger.

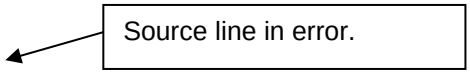
In this example, a script **Abort=** was set by an **AssignInit=** section that ensured there was at least one matching Support Organisation.

Another example where a bad value is passed as a script argument:



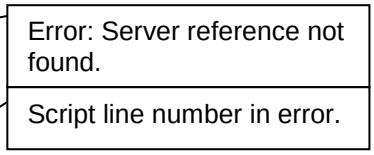

```
E:\> SthMupd -v -e FD-SupGrp-Ren.ini Do -Org "Qelp Desk"

Meta-Update    Version 5.56 (x64) for ARS lib 8.1.2
           (c) Copyright 1996-2015 by Software Tool House Inc.
               www.softwaretoolhouse.com
114159.515 q [Do] QuerySql: Svr: sthv1
114159.515 q [Do] QuerySql: Qualification: : 0000: select count(*)
             CTM_Support_Group where  Support_Organiza
114159.515 q [Do] QuerySql: Qualification: : 0040: tion = 'Qelp Des
114159.515 q [Do] QuerySql: returned 1 records of 1.
114159.515 i [Do] Msg: Found 0 records with:  'Support Organization
114159.515 E [Do] Msg: The Support Organisation argument must match 1 or more records
             of CTM:Support Group"
114159.515 E [Do] Msg: Please check the spelling of your command line argument."
114159.531 E [Do] Abort: ..aborting."
114159.531 E [Do] [asg-init] AssignInit apply was aborted in file:  FD-SupGrp-Ren.ini
             [asg-init]   @Cmd=  line: 74
114159.531 E IniRdo of FD-SupGrp-Ren.ini [Do] failed with 3 - ArPutIini: Parm error 3
114159.531 i Statistics:
114159.531 i         Sections:                    1
114159.531 i         Maximum section depth:       1
114159.531 i         Output Schemas:              0
114159.531 i         Output Schema records:       0   created
114159.531 i         Output Schema records:       0   updated  (with 0 skipped)
114159.546 i         Outputs OK:                  0
114159.546 i         Outputs Errors:              0
114159.546 i         Outputs Aborts:              0
114159.546 i         Input   Errors:              0
114159.546 E error: some errors occurred.  Check for errors above this message.
114159.546 E terminating unsuccessfully in 0 sec.
```

In this next example, the script file’s **Query=** at line 65 referenced a ReadServer tag which was not defined as the script didn’t need use additional servers.




```
     Query      = @Itsm6, User, User, $V, Qry$
```

```
E:\> SthMupd  QQQ-TblRpt-User.ini Do sthv1 Demo -start 1 –max 10
```




```
Meta-Update    Version 5.56 (x64) for ARS lib 8.1.2
           (c) Copyright 1996-2015 by Software Tool House Inc.
               www.softwaretoolhouse.com
113416.785 i [Do] One:
113416.785 i [Do] One: Launching:  1 of  1 [Do1]
113416.785 E [Do] One: FlIniFindCtl: Server Tag: Itsm6 not found
113416.785 E [Do] One: ArIiniQuery: FlIniRefFindCtl for Itsm6 failed at file:  QQQ-
             TblRpt-User.ini  [Do1]   Query=  line: 65
113416.785 E [Do] One: ArPutIiniRinit: ArIiniQuery failed (rc=4) in file:  QQQ-TblRpt-
             User.ini  [Do1]   Query=  line: 65
113416.785 E [Do] One: ArPutIiniRinit for Do1 returned 3 - ArPutIini: Parm error 3
113416.785 E [Do] One: ArPutIiniRdo: DoLaunch failed!
113416.801 E [Do] One: 0 record OK; 1 records with errors; total: 1.
113416.801 E IniRdo of QQQ-TblRpt-User.ini [Do] failed with 3 -
113416.801 i Statistics:
113416.801 i         Sections:                    1
113416.801 i         Maximum section depth:       1
113416.801 i         Singleton Sections:          1   errors:      0
113416.801 i         Output Schemas:              0
113416.801 i         Output Schema records:       0   created
113416.801 i         Output Schema records:       0   updated  (with 0 skipped)
113416.801 i         Outputs OK:                  0
113416.817 i         Outputs Errors:              0
113416.817 i         Outputs Aborts:              0
113416.817 i         Input   Errors:              0
113416.817 E error: some errors occurred.  Check for errors above this message.
113416.817 E terminating unsuccessfully in 0 sec.
```

Script Debugging

Software Tool House Inc

Script Debugging

What Is Script Debugging?

When running Meta-Update in debugging mode, you can

- View your script's source lines
- Set and manage breakpoints
- View references
- View help on the debugging commands available.

When running Meta-Update with the debug switch, Meta-Update will load the script file and then present you with the debugging prompt.

You can then set and clear breakpoints, and begin or continue execution, or single step through the script.

The debugger is part of the Meta-Update binary and is available on all supported platforms.

A normal debugging session comprises setting various breakpoints within the script, continuing execution until the breakpoints are reached, examining references and field values, and then resuming or aborting script execution.

Additionally, a conditional Break command may be coded in an assignment section of a script that will cause a breakpoint when the condition is met and debugging is enabled.

Entering Debug Commands

At the Meta-Update debugger prompt, you enter debug commands by specifying the command, any command arguments, and then ending the command with a new line.

The last command entered of a subset of the commands) is automatically repeated when the enter key is pressed with no command entered.

If there is no such command saved, or an invalid command was entered, help text is printed outlining the available commands.

Further help is available by using the Help command and specifying the command name you want more help on.

All Meta-Update debug commands may be abbreviated.

```
C:\> SthMupd -v -g ArSchema-Rpt.ini Do
                  -fle "ArSch.csv"  -qry RE:%

Meta-Update    Version 5.56 (x64) for ARS lib 8.1.2
          (c) Copyright 1996-2015 by Software Tool House Inc.
              www.softwaretoolhouse.com
211900.140 q Server:     (  5) sthv1
211900.140 q User:       (  4) Demo
211900.140 q Port:            2501
       42: [Do]
  at:   [Do]  ln 42 Init

Mupd Dbg > help
The Meta-Update script debugger supports these commands:
  h   Help        Displays Help about commands
  bt  BackTrace   Print a Launch backtrace
  p   Print       Print tag set or single string
  s   Step        Execute next statement
  so  StepOut     Execute until section end
  n   Next        Execute Next statement
  c   Continue    Continue execution until Break Point
  b   Break       Manage Break Points
  l   List        List script source
  lf  Listfiles   List included script files
  q   Quit        Quit job execution

Enter "help command" for more Help on these commands.

  at:     bp 2:  [asg-I]  ln 37 Asg

Mupd Dbg > go
```

Meta-Update Line Numbers

The **`@include`** directive allows a single Meta-Update script to include other script files.

The format of “Line Numbers” displayed and used as input in the Meta-Update debugger is changed as a result. There are two formats of line numbers when a script uses the **`@include`** directive:

- A single combined line number,
- A file specific line number comprising the file index and line number within that file.

The combined number can be used as input and is listed along with the file specific number when listing source lines. It is the line number of the file that results when all **`@include`** directives are resolved.

The file specific number consists of two parts: the “file index” and that file’s line number – not taking into consideration any other files.

The ListFiles command will list all source files and their indexes.

About Meta-Update Break Points

A “Breakpoint” in the Meta-Update sense is either

- A control section name and an “event”
- An assignment section’s line number

A control section has the following breakable events:

`Init`	when a section is starting up
`Term`	when a section is completing
`IterInit`	before an iteration query is run
`IterNext`	before an iteration record is loaded
`IterTerm`	after the last iteration record is completed
`Launch`	before each Launch is evaluated
`Asg`	a line number at an assignment section

`Init` **`Init`** happens before a section is ready to perform its iteration query, and before any **`AssignInit`** assignments have been done.

`Term` **`Term`** happens when all iterations of the section have been processed, all termination assignments have been processed and the script is ready to return to the launching section.

`IterInit` Happens only once per section call. Happens after all the **`AssignInit`** assignments have been processed and before the iteration query (or open file etc) has been executed.

`IterNext` Happens once per iteration. Happens just after the next iteration record, row, field set, has been loaded and just before any **`AssignPre`** assignments are processed.

I

`Launch` Happens once per iterations. Happens after all **`AssignPre`** assignments are processed and before each **`Launch`** is evaluated.

Asg Assignment statements are line number based and not event based.

Each assignment statement in an assignment section can be stepped through, one at a time.

When specifying an assignment breakpoint, you simply specify the script’s line number that you wish to break at – before its assignment is processed.

Note that when you set the breakpoint, Meta-Update does not check that the specified line number is in an assignment section. Use the List command to verify your line numbers.

It is not possible to set breakpoints in lookup sections or file sections.

A normal begging session begins with setting various breakpoints and then continuing execution until one of those breakpoints is reached.

Debug Commands

This table lists the Meta-Update debug commands.

Command	Abbreviation	Notes
Help	**h**	Displays Help about commands
List	**l**	List script source
Listfiles	**lf**	List script source files
BackTrace	**bt**	Print a Launch back trace
Print	**p**	Print tag set or single string
Next	**n**	Execute Next statement
Continue	**c**	Continue execution until Break Point
Quit	**q**	Quit job execution
Break	**b**	Manage Break Points

List

Use **List** to print your scripts source lines or sections:

```
Mupd Dbg > help list
  The List command allows you to display your Meta-Update
     script's Source lines

  l   List          Lists up to 25 source script lines

      List    nnn              starting at line nnn
      List    nnn, mmm         starting in file nnn line mmm
      List    nnn  mmm         same as above
      List    *                list section names and line numbers
      List    Sec              list contents of a section
      List    Sec  Kwd         list only a single keyword and value

  Examples
    l   100                    will list the source script at line 100
    l   2   20                 will list source at line 20 in file 2
    l   asg-new-HPD            will list the complete [asg-new-HPD]
    l   asg-new-HPD  Status    will list the Status= assignment in

  at:   [Do]  ln 42 Init

Mupd Dbg >
```

List Files

Use **ListFiles** to print your scripts source file name and file indexes:

```
Mupd Dbg > help listfiles
  The ListFiles command lists all source files and file indexes of the
             source script

  lf  ListFiles     [n]        list file names and indexes

  When using multiple source files with the @include directive,
             each file is given an index number from 1 upwards.
             Line numbers are displayed and entered as either
             [ix, num] where ix is the file index, or a single
             line number, being the combined line number for the
             script with all included files folded in.

  The optional argument causes the single file name referenced by
             the given index to be listed.

  at:   [Do]  ln 42 Init

Mupd Dbg > listfiles
1 --> 100-Hpd.ini
2 --> 999-Includes\000-Jctl-Sync.ini
3 --> 999-Includes\100-CfgUpdFlag.ini

  at:   [ Cfg-asg ]    35 [1,   35] Asg

Mupd Dbg >
```

BackTrace

Use **BackTrace** to print the list of sections launched to the current execution position. From the Help **BackTrace** command:

```
Mupd Dbg > help bt
   A Launch BackTrace reports which sections have lead to
   where you are in a Meta-Update script.

  bt  BackTrace     Print a Launch BackTrace


  Examples
    bt
     . [DoOuter]  @ line 14
     . . [DoInner]  @ line 21
     . . . [DoInnerInner]  @ line 42
     @ DoInnerInner line 42

  at:   [Do]  ln 42 Init

Mupd Dbg >
```

Print

Use **Print** to print the available reference tags, all references for any one tag, or a string containing references. From the Help **Print** command:

```
Mupd Dbg > h print
  The Print command allows you to print your Meta-Update
     script's Tags and variables

  p   Print      Print tag, set, or single string

      Print                 will print all Tags defined
      Print      Tag        will print all members of the Tag
      Print      String     will print String with normal substitution

  Examples
    p                          will print all defined tags
    p    ENV                   will print all environment variables
    p    ArsSvr=$CTL, Server$
                                will print ArsSvr=xxx

  at:   [Do]  ln 42 Init

Mupd Dbg >
```

Next

Use Next to “single step” your script.

If you’re in an assignment section, next will execute the current assignment statement and then stop at the next one.

If you’re in a command section, next will run the next “phase” or operation in that command section and stop before the next operation. For example, next might load the next iteration record and then stop before executing any AssignPre= assignment sections.

```
Mupd Dbg > h next
  The Step command executes the next instruction or settable breakpoint
     and returns debugging control to you.

  n   Next                    Execute next instruction

  See also:
      Continue"

Mupd Dbg >
```

Continue

Use Continue to resume normal execution of your script until a breakpoint is reached or the end of the Meta-Update job is reached.

A normal begging session begins with setting various breakpoints and then continuing execution until one of those breakpoints is reached.

```
Mupd Dbg > h continue
  The Continue command continues script execution until the next break.
    point is reached.  Use break clear all to remove all breakpoints
    before entering the Continue command to run the script to its end.

  c   Continue         Continues script execution until the next
                       breakpoint is reached or until the end of
                       the Meta-Update job.

  See also:
      Quit

Mupd Dbg >
```

Quit

Use Quit to terminate the Meta-Update job immediately. An error message is written and the Meta-Update job ends abruptly.

A normal debugging session begins with setting various breakpoints and then continuing execution until one of those breakpoints is reached.

```
Mupd Dbg > h quit
  The Quit command terminates the Meta-Update job immediately.
     Use Continue to continue script execution until the next break.

  q   Quit             Terminates this Meta-Update job immediately
                       with an error

  See also:
      Continue

Mupd Dbg >
```

Break

Use Break to manage your script's breakpoints. With Break, you can set, list, or clear script breakpoints

```
Mupd Dbg > h break
  The Break command allows you set, clear, and list Break Points.

  Break Points allow your Meta-Update to proceed until you reach
  an area of the script you want to examine.

  A "Break Point", in the Meta-Update sense, is a section name and
    a Section's Event Type.  Event Types are things like before an
    Iteration Query is loaded, or after a new Iteration record is read.

  The Break command allows you set, clear, and list Break Points.

  b   Break                Manage Breakpoints
  bs  Break Set            Set a Breakpoint
  bl  Break List           List Breakpoints
  bc  Break Clear          Clear Breakpoints



  bs  Break Set            Set a Breakpoint

  bs  line                 Will set an Assignment break point to
                            the line specified
  bs  section  type
           section     is a section name in the script
           type        is one of:
                Init       when a section is starting up
                Term       when a section is completing
                IterInit   before an iteration query is run
                IterNext   before an iteration record is loaded
                IterTerm   after the last iteration record is completed
                Launch     before each Launch is evaluated
                Asg        at an assignment section; must be set
                            with bs line

  Examples:
    bs  42
                        breaks when line 42 is encountered while
                         processing an assignment section
    bs  Do  IterInit
                        breaks just before an iteration Query is run
    bs  Do  IterNext
                        breaks just after each iteration tag is loaded
    bs  Do  Launch
                        breaks just before each Launch of this Section

  See also:
      Next, Step, Continue

Mupd Dbg >
```

Script Reference

Script Reference

Script File: General Format

The script file drives Meta-Update. It is your Meta-Update script. It tells which form the target assignment is to be applied to, and drives the required loading of records.

It resembles a sectioned INI file:

```
[Main]
Server          =   ArsDev
User            =   Demo
ArgNm           =   HpdId

[Controls]
Update          =   Tgt, HPD:HelpDesk,          &
                    '1' = "$Arg, HpdId$"$001
Assign          =   Assignment


[Assignment]
Description     =   "Router "
Description     =   " down; auto-raised by Xxx"
Summary         =   "Auto"
Status          =   Assigned
```

The [Main] section identifies the ARS server and sign on parameters.

The [Controls] section is passed on the Meta-Update command gives operational information including the Assignment section to be applied.

The [Assignment] section assigns values to fields in HPD:HelpDesk for update.

The format for this INI file is as follows:

- Comments may be coded freely. They are started with a number sign ("#") or semi-colon (";") as the first non-white space character of a line. Blank lines may also be inserted freely. Comments cannot be coded on the right side of lines as the '#' character is permissible in ARS form names, field names, and queries.

- Lines can be continued by having the last non-blank character of a continued line be a backslash ("\") or ampersand character ("&").
  - If a backslash is used, all spaces preceding the continuation character and at the beginning of the next line are significant. No additional spaces are inserted.
  - If an ampersand is used, all spaces preceding the continuation character and all leading spaces on the continuation line are removed and a single space is inserted.

- The @include file directive will include the whole of another script file and then continue reading the source file at the same point. The resulting script is a merge of all source script files.

- All section names and keywords are case sensitive.
- Keywords within a section can be placed in any order but are processed in the order that they are encountered.
- Sections can be placed in any order and can be split.
- Equal signs only are used to separate a keyword from its value.

- The file may be in either Windows or UNIX formats. That is, lines may be terminated by either <lf> or <cr><lf> , and a single end of file marker (^Z) will be ignored if present as the last character of the file. Script files may be used across both platforms.

The validity of the INI file can be checked with the siniget.exe program. This is highly recommended whenever the INI file is changed, as all invocations of Meta-Update specifying that file will fail if its syntax is in error. To check the syntax, simply invoke the siniget.exe executable with the INI file name as the only parameter. It will either report a syntax error, or it will print the contents of the file.

Including Other Script Files

Script files may include other script files. The resulting script that Meta-Update executes will be the merge of all source files and included files in the order that they are included.

The @include file directive tells Meta-Update to stop processing this script source and fold in the included source, and finally to resume processing the original script source at the line after the include statement at the original source's section.

The file name specified on the directive cannot be a reference and must be a valid file name for the operating system on which Meta-Update is running. For example, directory separators must be the correct ones for Linux and Windows.

The **`SthScriptPath`** environment variable may be used to set a search path for the included files. Then the script can simply reference the file name with no path information. That script can then be used on Linux and Windows.

An example may help:

```
include     main.ini

[Controls]
@include    controls.ini

[Assignment]

Summary     =     "Auto"
Status      =     Assigned
@include    asg-desc.ini
```

```
[Main]
Update   =   Tgt, HPD:HelpDesk,   &
             '1' = "$Arg, HpdId$"
Assign   =   Assignment
```

```
Description = "Router down; "
Description = "auto-raised by Xxx"
```

The script above is entirely the same as the script below.

```
[Main]
Server      =     ArsDev
User        =     Demo
Arg         =     HpdId

[Controls]
Update      =     Tgt, HPD:HelpDesk,   '1' = "$Arg, HpdId$"
Assign      =     Assignment

[Assignment]
Description =     "Router down; "
Description =     "auto-raised by Xxx"
Summary     =     "Auto"
Status  =   Assigned
```

Section Types

There are several types of sections used in the ini file. These are:

Main Read Servers	gives the updating ARS server and sign on information. gives ARS server and sign on information.
Control	specifies the operation you want Meta-Update to perform. You can iterate through a query or loop and output files and ARS records.
File	defines the format of an external ASCII file that will either be read or written.
Field	defines a set of fields to be associated with an external ASCII file, an SQL result set, a regular expression pattern. Used to specify value transformations and interpretations.
Assignment	contains the actual field assignments to be made to the target form. These can include other assignment sections and can reference Look Up sections.
Look Up	offers a non-Remedy mechanism for translating data values using lists, CSV files, ARS Queries, SQL queries or procedures.

When you fire Meta-Update, you pass it a single Control section's name.

This control section also lists any assignment sections to be applied to the target update record, and other control sections through the Launch statement.

Think of this section as a "main" or "entry point" to a script. A script can be coded with multiple starting sections. Consider, for example, these "entry points in the same script, that either iterate or not, and then all launch the real worked section:

```
Ppl-Del.ini          Del-One       -p     hwu
Ppl-Del.ini          Del-File      -p     del-list-42.csv
Ppl-Del.ini          Del-Qry       -p     "'1' = 42"
```

Only those sections following within the run of a Meta-Update script are syntactically checked. For example, if the first control section launches a second control section conditionally, and that condition is not met, then that second section will not be syntax checked as it was not fired.

[Main] Section

The [Main] section is required and declares the Remedy sign-on information, if not entered on the command line. This example is used by the samples.

```
[Main]
Server      =     $ ENV,  ArsSvrAdmin $
Port        =     $ ENV,  ArsPort     $
User        =     $ ENV,  ArsUsr      $
Password    =     $ ENV,  ArsPwd      $
```

The above environment variables are set by a Windows batch file, `SthLic.cmd`, or a Unix shell script, `SthLic.sh`. This batch file sets the environment variables in the current Window for any given Meta-Update licensed BMC Remedy server. It also sets the license variables for all Meta-Update utilities.

`SthLic.cmd`, or, `SthLic.sh` are generated by a utility called **SthLicUpd**, or the Software Tool House License Updater. See "Running SthLicUpd above for more information.

The following keywords are available in **[Main]**

ARS Authentication	
`Server`	Specifies the server connection address. May be a reference. Must resolve into an IP address that has an BMC Remedy Server listening for API requests.
`Port`	Specifies the BMC Remedy Server's Listen port. Use zero when the server uses Port Mapper.
`RPC`	Specifies the BMC Remedy Server's RPC Listen port. Not used by Meta-Update.
`User`	The BMC Remedy server Login Name to be used for the script. It is highly recommended that this user be an administrator and have all ITSM rights. Some script operations, such as **`QuerySql=`**, require Administrator rights. When a non-administrator is used, it is possible for scripts to be denied fields, records, or operations.
`Password`	The above user's password. May be an encrypted string as
ARS Session Control	
`TimeOutNormal`	Specifies the "Normal" time-out value. Default and minimum is 120 seconds. Primarily used for reads and updates. May be a reference.
`TimeOutLong`	Specifies the "Long" time-out value. Default and minimum is 300 seconds. Primarily used for queries. May be a reference.
`Locale`	Specifies the BMC Remedy Server's RPC Listen port. Not used by Meta-Update.
`ClientType`	The BMC Remedy server Login Name to be used for the script. It is highly recommended that this user be an administrator and have all ITSM rights. Some script operations, such as **`QuerySql=`**, require Administrator rights. When a non-administrator is used, it is possible for scripts to be denied fields, records, or operations.

Password	The above user's password. May be an encrypted string starting with "Enc:" or a plain text password. Use '-' for no password.
Meta-Update Licensing	
License	The License key may be specified in the script. This is not recommended as there are other, more convenient ways of specifying the license key, including by using the Windows batch file, **SthLic.cmd**, or the Unix shell script, **SthLic.sh**.
Meta-Update Scripting Options	
MaxOutput	**nnn** Default: none Can be used while developing scripts. Limits the number of Output records. This includes Updates and Creates.
QueryDbNames	**Yes, No** Default: Yes Disallow use of BMC Remedy field database names in queries.
IdLog	**filename** Default: none Specifies a default, script-wide IdLog. This has been superseded by the section based IdLog facility. Please see Using IdLogs in for more information.
Script Arguments	
Arg Arg	**Arg name [Default xxx]** Default: none **Arg name [Default xxx]** Specifies a "field" that will be referenced in the script body in the tag "Arg". This will correspond to an argument on the command line when this script was invoked. If the argument default value is not specified in the script, the argument will be required on the command line and an error will be thrown. Any script usage text (**PrmReq=**) will be displayed. You may specify as many **Arg=** keywords as needed by the script. It is recommended that a minimum of one argument be mandatory and that script usage information be supplied with the **PrmReq=** keyword.
ArgNm	**ArgNm=** is deprecated and is treated as **Arg=** The **Default** keyword is not available..
PrmReq PrmReq	**text** Default: 0 **text** Specifies usage text presented when any required arguments are not supplied. All sample scripts will present usage instructions when run without arguments. These will explain the script function, default and required arguments, and example runs.

Script Initialization	
ReadServers	**Section, section,** … Default: none A list of other **ReadServer** section names which will be used in the script. See **ReadServer** sections below. Sessions will be established to the main server and all servers in the sections specified by this statement. Queries and SQL queries can be run against the main server or any of the Read Servers.
RandSeed	Yes | No Default: Yes If set "No" the sequence of random numbers generated by the rand() function will be the same on multiple runs.
AssignInit	**Section, section,** … Default: none A list of Assignment sections that are invoked before the first control section passed on the command line begins. This happens after arguments are processed and the **[Main]** and all **ReadServer** sessions are established. It is effectively the first assignment section. See **AssignInit** below and in the Assignment reference.
AssignTerm	**Section, section,** … Default: none A list of Assignment sections that are invoked after the first control section passed on the command line completes. This happens just before the job ends. See **AssignInit** below and in the Assignment reference.

`Server =` This is the Remedy ARS server to sign in to. If coded on the command line, this has no effect. If not coded on the command line, this must be coded. This does not have to be the real Remedy server alias. It is simply an IP or domain name that translates to an IP where the ARS server is running. The server may be specified as a string reference. The string reference may be a named parameter or an environment variable. If it is a named argument, it must be passed on the command line as a script argument.

`User =` The ARS User ID to use if not coded on the command line. This may be a string reference.

`Password =` The password for the above user. Code "-" if there is no password.

Use the operating system security to prevent unauthorised access to the file. This may also be a string reference such as in the default value:

```
$ ENV, ArsPwd $.
```

The ARS User Password may be encrypted. If so, it begins with "`Enc:`". Password encryption is handled by a separate utility, **SthLicUpd**. This utility is used to both generate the **SthLic.cmd** file and to encrypt ARS User passwords.

`Port =` This is generally not specified. It is required if the ARS server does not use Port Mapper. Simply code the port that the ARS server uses. Note that if the environment variable, ARTCPPORT is set, this setting is ignored. This is documented in the ARS manuals. This may be a string reference. Setting this to zero (0) has the same effect as not specifying it at all.

`RPC =` This is generally not specified. It is required if the Meta-Update process is to use a private queue on the ARS Server. Simply code the RPC program number that the ARS server has been configured to use for Meta-Update. Note that if the environment variable, ARRPC is set, this setting is ignored. This is documented in the ARS manuals. This may be a string reference.

This ARS server must be licensed for Meta-Update use.

Records will be updated on this server.

Other servers may be read from. These are called ReadServers. These do not need licenses.

`MaxOutput =` Optional. Limits the number of outputs for the entire job to a specified maximum. Any single record Updates, Creates, File writes is considered in this maximum. The default is 0 which means unlimited.

This is useful during development of scripts that return large query results or process large files.

`RandSeed =` Optional. Meta-Update, by default, seeds the standard random number generator with the run time at start-up. You can have Meta-Update not seed the random number generator by specifying RandSeed = No.

Note that the sequence of random numbers generated on each run will always be the same for a script that does not seed the generator.

`QueryDbNames =`
Optional. Meta-Update, by default, allows a field's database name to be used within single quotes in a Query. Meta-Update pre-scans all Query qualifications and if it sees a field's database name will substitute the field Id within the single quotes. This is especially useful as field labels can change when multiple languages are installed and a forms default view is changed.

By specifying QueryDbNames=No there will be no prior scan of qualifications.

`TimeOutNormal =`
Optional. Can be used to increase the "Normal" timeout value for this session. Only available for ARS Release 6.3 or above. The default or minimum is 120 seconds. This value applies to record reads and submits. If a lot of workflow is run when a record is submitted, raising this value may correct the problem. Symptoms of the problem are an ARS error 92.

`TimeOutLong =`
Optional. Can be used to increase the "Long" timeout value for this session. Only available for ARS Release 6.3 or above. The default or minimum is 300 seconds. This value applies to record queries. If slow queries are run through ARS, raising this value may correct the problem. Symptoms of the problem are an ARS error 93 or 94.

`Locale =`

Optional. Used to set the ARS server's client locale for the RPC calls in this Meta-Update execution. Only available for ARS Release 6.3 or above. The default is "" or "C".

The Remedy API uses this client Locale setting to effect character translation to and from the internal database representation and to interpret field labels in queries. Meta-Update does not validate this setting.

Please see the BMC Remedy Installation and Configuration manual for more information on the values that can be used in this setting.

`ReadServers=` Optional. Specifies one or more section names defining additional ARS servers and the Tags associated with them. These servers can be queried and read but not written to. These servers do not need Meta-Update licenses.

`IdLog =` Deprecated and superseded by section IdLogs which allow more functionality such as assignments, fields, conditions.

Optional. Specifies a file name to be produced. This file will be a tab separated columnar file containing a single header row and a row for every record added or updated in the Meta-Update run and every record queried or loaded from a file/.

```
IdLog  =      fname   [ , { Overwrite | Append } ]
```

The file name can use substitution from parameters and environment variables and is in the form of a string reference (see below).

One of the two keywords, "Overwrite", or "Append" can be coded following the filename. The default is "Append".

The produced file can be imported into Excel and looks like this:

Time Server User Schema ID Op Op2 Status

The records are produced in the order that the queries and updates are done. Time is only resolved to a second.

On updates and ARS queries, the full Schema name is identified and the ID of the record is specified (unless a create operation failed).

On updates of Join forms, the record id is blank. Note that on a Join form, it is the workflow that creates underlying records when desired. A submission to a join form with no workflow defined succeeds but causes no database updates.

In the case of a file, there is no User, the Server is the file name, the ID is the record number, the Schema is, by default, the first 20 bytes of the record itself. This value may be changed when defining the file.

Op contains either "Update", "Create", or "Read"
Op2 contains "Merge" if and only if a Merge operation was done.
Status can be one of
- Ok the operation completed successfully
- Ok – Skip the update was skipped as no fields had changed values
- Error the operation failed

`PrmReq =` Optional. If coded, specifies script usage text that will be produced as error messages if required script arguments were not coded on the command line.

This is generally a good place to put usage information about the file.

`ArgNmVar =` Deprecated. There is no need to use this.
Optional. If coded, specifies a Reference name to be used for command arguments (parameters). If missing, the name is presumed to be "Arg".

`Arg =` Optional. If coded, specifies a command argument and optionally a default value. Only arguments without default values are required on the command line.

```
Arg = arg_name          Default "default"
```

The reference would become **Arg, arg_name**

For the switch based command line, the argument name is used as a switch and it is followed by the argument value:

```
SthMupd.exe MyScript.ini Do –arg_name "some value"
```

`AssignInit =` Optional. If coded, specifies a list of Assignment Section names to be processed after Meta-Update establishes all its server sessions and before the first Control section is fired.

`AssignTerm =` Optional. If coded, specifies a list of Assignment Section names to be processed after Meta-Update completes all script processing and before the server sessions are closed.

You may specify script-wide initialization and termination assignments from **[Main]**

These assignment sections can be used to load records, load configuration values, validate the environment, fire processes, and so on.

Please see the Assignment reference below for more information

The Meta-Update License may be specified in the Main section of a command file as an alternative to using environment variables or using a form on the server. There are two types of licenses: Server and Site.

`Site =` This is the name the Site for a site license. It must be specified exactly as was specified when the site license was requested.

`Domain =` This is the Domain suffix for a site license. It must be specified exactly as was specified when the site license was requested.

`License =` This is the password for either a server or a site license. It must be specified exactly, as was specified when the license was requested. If a site license is being specified, both the Site= and Domain= will be required.

Examples

```
[Main]
PrmReq          =       Usage     . . .     -TtId  TT-ID    - PplId  PPL-ID
Arg             =       TtId
Arg             =       PplId
```

In the above example, in subsequent references, the TT-ID parameter may be referenced in an assignment statement:

```
Xxx     =       Arg, TtId
```

Or an expression:

```
Xxx     =       @if(”$Arg, TtId$” = ”All”, ”%”, ”$Arg, TtId$”)
```

Read Server Sections

Read server sections identify additional ARS Servers that can be queried or read from.

A Tag or name is specified to identify the server. All read server sections are identified on the `ReadServers=` entry in the `[Main]` section.

```
[Main]
ReadServers   =      ReadSvr1, ReadSvr2

[ReadSvr1]
Tag           =      Svr1
Server        =      198.2.12.1
Port                 $Arg, Svr1port$
RPC                  $Arg, Svr1rpc$
User          =      Demo
Password      =      xxx
TimeOutNormal=       240
TimeOutLong   =      600

[ReadSvr2]
Tag           =      Svr2
Server        =      198.2.12.2
User          =      Demo
Password      =      xxx
TimeOutNormal=       360
TimeOutLong   =      1200
```

The `Tag=` specifies the word used to identify the ARS Servers in the control section's `Query=` or `Load=` statements.

Read Servers do not need to be licensed. Sessions are established for each ReadServer section specified in the [Main] ReadServers= values.

Like the main section, values for Server, Port, RPC, User, and Password may be string references. Values may also be set for time-outs and for Client Type.

Control Sections

About Control Sections

A Meta-Update Control Section tells Meta-Update the operations to perform.

When you fire Meta-Update, you pass it the first control section's name. You may code many sections in the same file.

A control section may execute its process once or may ***loop*** through the

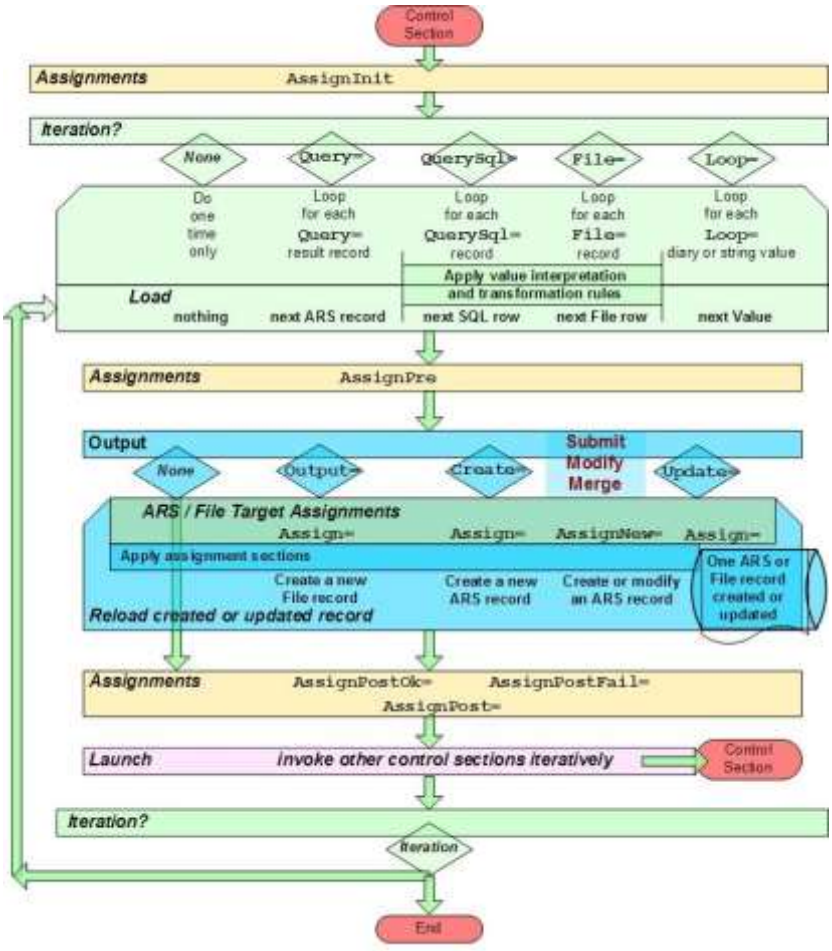
- records returned by an ARS Query or an SQL Query,
- records read from an ASCII file,
- values extracted from a string or a Diary field,
- the fields of a record,
- while any condition remains true.

This is called ***Iteration***.

A control section can ***`Create`*** or ***`Update`*** ARS records or file records and can ***`Launch`*** other control sections to create or update more ARS or file records.

A control section tells what output, if any, to produce: the target form (schema), whether this will be updating or submitting new records, whether this will use the normal Submit or Modify API or use Merge.

It also gives the ***assignment*** section names to be applied to the target update record and lists any assignment sections to be applied when the control section starts, ends, or before or after the iteration record is loaded.



A control section can also ***`Launch`***, or call other control sections in order and conditionally. These can process queries dependent on this query's retrieved records or the first section's record and can create other outputs and launch other sections as needed.

A control section can also have no output at all. It can be used to group several control sections or decide which control sections to launch based on arguments or other criteria.

Keywords & Statements

A control section can use these types of statements to:

- Operational — control Meta-Update's behaviours.
- Load — allow loading of additional records. This is superseded by the LookUp facility and use is discouraged.
- Iteration — automatically iterate through the rows of a Query, QuerySql, File, values extracted from a string or diary field, or on any condition.
- Output — update an ARS record or add a row to an output file.
- Launch — call another Meta-Update control section to query and update more records.
- Assignment — specify assignment sections to be called at various points in the cycle of a control section.
- IdLog — specify an "IdLog" to automatically create CSVs on a section's events.

This table specifies all Control Section statements:

Operational statements	
`Sleep`	Used to slow down the operation of Meta-Update.
`Status`	Alters or inhibits the default status message while processing a file or query.
`TimeOutNormal`	Alters the ARS Defined "Normal Operation" timeout value.
`TimeOutLong`	Alters the ARS Defined "Long Operation" timeout value.
Load statements	
`LoadQ`	Specifies a query that results in a single record to be loaded.
Iteration statements	
`File`	Indicates that this operation will process an ASCII file. Point to the file definition section.
`Query`	Indicates that this operation will process a set of records returned by a query.
`QuerySql`	Indicates that this operation will process a set of direct SQL records returned by a query.
`Loop`	Indicates that this operation will process a string or diary field value, loop through the fields of a Tag or the forms making up a Join, or while a condition is true.
Iteration controlling statements	
`Until`	Applies a conditional expression to limit the number of iterations the section performs.
Output statements	
`Update`	Specifies that an output record is to be created or updated. Specified the query to be issued to determine the update record.
`Create`	Specifies that an output record is to be created.
`Output`	Specifies that an ASCII file record or pattern file is to be utput

	rather than an ARS record.
`Assign`	Specifies the assignment sections to be applied to an update record.
`AssignNew`	Specifies the assignment sections to be applied if an Update= query returned zero records and you want to create a new record.
`AssignOpen`	Specifies the assignment sections to be applied if an Output= file is specified as output. These sections are applied only when the file is opened.
`AssignClose`	Specifies the assignment sections to be applied if an Output= file is specified as output. These sections are applied only when the file is closed.
Output controlling statements	
`UpdateIfEqual`	Specifies whether to continue with an update when no field values have been changed.
`Merge`	Indicates whether a Merge operation is desired and specifies the Merge options desired. Ignored if the output is not an ARS record.
`MaxOutput`	Limits the number of outputs for this control section to a specified maximum. Any single record Updates, Creates, File writes is considered in this maximum. The default is 0 which means unlimited. This is useful during development of scripts that return large query results or process large files. Note that when coded in the `[Main]` section, any run of this section is limited to the lesser of the two numbers.
Launch statement	
`Launch`	Specifies a set of Control sections to be fired for each record updated in this operation. May be conditional. Used to Nest operations.
Assignment section statements	
`AssignInit`	Specifies the assignment sections to be applied before processing begins for the section.
`AssignTerm`	Specifies the assignment sections to be applied after processing ends for the section and the section is about to be closed.
`AssignPre`	Specifies the assignment sections to be applied directly after loading the section's record. This is applied before any subsequent Loads, Updates, or Assignments.
`AssignPost`	Specifies the assignment sections to be applied directly after completing any updates but before any launches and the next iteration of the section. This is applied whether an error occurred or not.
`AssignPostOk`	Specifies the assignment sections to be applied directly after completing any updates but before any launches and the next iteration of the section. This is applied whether only when an error did not occur in the update or any launches. .This is applied before the AssignInitPost sections.

AssignPostErr	Specifies the assignment sections to be applied directly after completing any updates but before any launches and before the next iteration of the section. This is applied whether only when an error did occur in either the update or any launches. .This is applied before the AssignInitPost sections.
AssignPostLaunch	Specifies the assignment sections to be applied directly after all launches are invoked, if any, but before the next iteration of the section. This is applied whether only when an error did occur in either the update or any launches. .T
IdLog statements	
IdLog	Specifies an IdLog file for a set of events and conditions. Also specifies the formats and assignments for an IdLog file.

Operational Statements

`Sleep =` `s [ , n ]`

Optional. If coded, specifies a number of seconds to pause every "n" iterations of a section. If "n" is missing, it defaults to 1.

Examples:

```
Sleep = 2
Sleep = 2, 5
```

The first example pauses 2 seconds after each iteration of the section. The second example pauses 2 seconds once after every five iterations of a section.

`TimeOutNormal =` `nnn`

Optional. If coded, sets the API Session Timeout for "Normal Operations" to a number of seconds. These operations include most single record operations such as reading and updating Remedy ARS records.

The number can only be increased from the default: 120 seconds.

Example:

TimeOutNormal `= 300`

This increases the timeout for single record operations to 5 minutes. This can be used when filter activity takes more than the default of two minutes or for slow connections to the server.

`TimeOutLong = nnn`

Optional. If coded, sets the API Session Timeout for "Long Operations" to a number of seconds. Long operations are those that are multi-record such as ARS or SQL queries.

The number can only be increased from the default: 300 seconds.

Example:

TimeOutLong `= 1200`

This increases the timeout for queries to 20 minutes.

`ClientType = nnn`

Optional. If coded, sets the Client Type. Setting the Client Type can be used in testing scripts to exercise client specific filters.

Example:

ClientType `= 11`

Set the Client Type to that of the Mid-Tier.

Load Statements

The **`LoadQ`** statement is supported but has been superseded by the more powerful LookUp facility.

The LookUp facility can cache records, handle multiple records as a result of a query and succeed even if no record is loaded.

A LoadQ is used to query for and load a single record into the specified Tag.

`LoadQ =` Optional. Loads specify a query that must return exactly one record.

```
LoadQ    =    [ @SvrTag ] Tag , Schema , Query
```

Any number of load statements may be in the control section or in the assignment section. These are processed in the order they are encountered with those in the control section being processed before those in the assignment section.

All loads coded in the control section before the iteration statement (`Query=, File=, Loop=, QuerySql=`) are processed before the iteration statement, if coded.

The iteration statement may reference fields from any of the preceding Load statements.

Loads following the iteration statement can refer to data from the loaded records or from the record loaded by the iteration statement.

Then, the `Update=` is processed possibly resulting in another load. The `Update=` can refer to any loaded record in the control section including the record from the file or query.

Finally, the assignment sections are processed. In any one assignment section, loads are processed first, then, an update record is built up. After all the sections are processed, the update is applied.

About Iteration Statements

Exactly one or zero iteration statements may be coded. If none are coded, the section performs its process exactly once, producing a single output, if coded.

If an iteration statement is coded, the section loops based on the results of the iteration statement, loading the values into the specified tag and producing its output, if coded, as many times as it iterates.

Query = Optional. A single `Query=` statement is allowed. If coded, a query is executed and the records returned from the query are iterated through.

They are loaded one by one and the assignment sections are applied to a new or retrieved update record. As many records as there are returned from the query are produced for the target schema.

```
Query    =   [@SvrTag                                    &
             Tag,                                        &
             Schema,                                     &
             [@sort(Fld [Ascending|Descending] [ , …]),]  &
             Query
```

`@SvrTag`	If coded, specifies that the Query is to be run against the specified Read Server.
`Tag`	As each record is loaded, references to the record's fields are made with this Tag.
`Schema`	This is the name of the ARS form to query.
`@sort`	Specifies a sort order. Records are normally retrieved in the sort order specified by the form definition with the admin tool. The default sort order is by Request ID which is generally from oldest to newest.
`Query`	This is an ARS query to be performed. The Query format is the same as that which is acceptable in the Advanced Query bar in the BMC User Tool. That is, field labels and not database names are used. Field Ids can be used when labels are not available of are multiply defined in the form. Of course, full reference substitution is available.

Examples:

```
Query = @Prod,                                          &
        SrcTT,                                          &
        HPD:HelpDesk,                                   &
        '1' = "$Arg, TT-ID$"

Query = SrcTT,                                          &
        HPD:Help Desk,                                  &
        'Assignee+' = "$SrcTT, Assignee$"  AND          &
        'Status'    < "Resolved"
```

QuerySql = Optional. A single `QuerySql=` statement is allowed. If coded, a query is executed and the records returned from the query are iterated through.

The returned SQL rows are loaded one by one into the Tag specified, and the assignment sections are applied to a new or retrieved update record. As

many records as there are returned from the query are produced for the target schema.

The `QuerySql=` returns a set of record with each record containing a set of fields. These fields can be referenced by either an integer or a field name. The field name is defined is a special Field section. This also allows data conversions to be specified from the native SQL data types into the ARS types. See Field Sections below for more information on specifying field value transformations.

The `QuerySql=` may be run on the target server or on any read servers. It is passed through the Remedy API to the Remedy server and executes the query using the Remedy server's credentials. There is size limit on the query itself.

```
QuerySql    = [ @SvrTag ] Tag , FieldSec,  Query
```

Loop = Optional. A single `Loop=` statement is allowed.
Loops can go through:

- Diary fields – assigning various fields from the Diary entry to the tag
- Delimited Strings – assigning the single string to the tag
- Fields of a schema or tag – assigning information and value fields to the tag
- Forms making up a Join – assigning form information to the tag
- As long as a while condition is true – not using a tag at all

These values are loaded one by one and the assignment sections are applied to a new or retrieved update record. As many records as there are values in the passed string or diary field value are produced for the target schema.

If the string or diary field value is null, no records will be produced.

```
Loop     =   Diary,  Tag , [ Reverse, | Forward, ]  Reference
Loop     =   String, Tag , seperator,                        &
             [ Reverse, | Forward, ]                         &
             Reference
Loop     =   Fields, Tag , Source Tag
Loop     =   Join, Tag , "Form Name"
Loop     =   While, (condition)
```

`Merge` = Indicates that a Merge operation is desired and specifies the Merge options desired.

Note that a Merge operation is different than a Submit or Update. A Merge is what the arimport facility uses. On Merges:

1. only workflow set on Merge will be fired – unless the `NoFilters` option is specified.
2. Core fields can be assigned or updated including the ID field.
3. Diary fields can be replaced completely with formatted Diary values; simple character strings are invalid as a Diary value.

```
Merge    =          Yes
Merge    =          [No]AllowNull , [No]SkipPatternMatch
                    [No]Filters
```

`AllowNull` Allows $NULL$ assignments to required fields

`SkipPatternMatch`	Allows assignments to fields even if the assignment fails the field's pattern matching specifications
`NoFilters`	is only available in Meta-Update for version 6.3+ of ARS. It causes all Merge filters to be turned off for the single Meta-Update job running.

The defaults are `NoAllowNull, NoSkipPatternMatch,` and `Filters.`

If the defaults are required, you can simply specify "Yes" to tell Meta-Update the Merge itself is required. For documentation and completeness, it is recommended that all options always be specified.

`UpdateIfEqual =`

The default action of Meta-Update is to skip a record update if the values being assigned are equal to the current database values. This can be used to override this default. If `UpdateIfEqual = Yes` is coded, an update will occur whether or not the values being assigned differ from the current values. This is useful for causing filters set on Modify to fire.

```
UpdateIfEqual    =          Yes | No
```

`Update =` Indicates that this is an update and supplies the query to be used to determine the update record. It must not be coded for creates.

```
Update    =  Tag , Schema , Query
```

If the section contains a `Query=` and the query results are to be updated, the `Update=` specifies the same Tag as on the `Query=`. The `Query=` cannot have specified a read server.

```
Update    =  Tag
```

A query is be performed to select the update record unless the update record is the same as this sections query record and the short form of the update statement is used.

This `Update=` query's results must contain exactly one or zero records. The Tag must be unique and cannot match that of the `Query=` if a different schema and query are coded.

If the `Update=` query returns zero records, a new record can be created if the `AssignNew=` is also coded. Otherwise, an error will be produced and no record will be created.

If the `Update=` query returns one record, and no `Assign=` is coded, no update will take place and no error will be thrown.

`Assign =` Required. Specifies the name(s) of the assignment section(s) to be applied to the updating record when `Update=` is used, or record being created if `Create=` is used.

These are the actual Remedy ARS field assignments to be performed against the target schema in either an update or a submit. See Assignment Section below for more information. Multiple assignment sections can be specified on multiple `Assign=` statements. All are processed in the order specified for each update.

`AssignNew =` If an `AssignNew=` is coded when an `Update` is used, and the update query results in zero records, this indicates that a new record is to be created. It lists the assignment sections to be applied for this condition. If it is not coded, no update is done when the update query returns no records.

```
AssignInit    =
AssignTerm    =
AssignPre     =
AssignPostOk  =
AssignPostErr =
AssignPost    =
```

The above keywords specify optional assignment sections. The different keywords indicate when, during the execution of a single command section, the assignments will be processed.

These are used in more complex scripts. Assignment sections so specified have no Remedy targets and are generally used to set script variables, or, launch external processes.

Only the following assignments can be made in these sections:

```
@Cmd = Reference
@Cmd = @if, else, endif
@Cmd = Include
@Cmd = Spawn
@Cmd = Abort
```

`AssignInit=` Specifies the name(s) of the assignment section(s) to be applied when a command section first starts. This is generally used to assign initial values to variables.

When a section is launched iteratively, each new Launch will process these assignments.

`AssignTerm=` Specifies the name(s) of the assignment section(s) to be applied once just before the section is ended. If a section is launched iteratively, then each time the section completes and is ready to return to the section will have these assignments processed.

`AssignPre=` Specifies the name(s) of the assignment section(s) to be applied before the next iteration of any Query= or File= statements is processed but after the Query= or File= record is loaded and a Status= message is processed.

If a section is launched and has no Query= or File= then this will have the same effect as an AssignInit.

`AssignPostOk =`

Specifies the name(s) of the assignment section(s) to be applied after an iteration of the command section is complete. That is after the update is done and all launches have completed. These assignments are applied only if the update and all launches succeeded. They are applied before any AssignPost or AssignTerm assignments.

`AssignPostErr =`

Specifies the name(s) of the assignment section(s) to be applied after an iteration of the command section has complete with an error. These assignments are applied only if the update fails or any of the launches fail. They are applied before any AssignPost or AssignTerm assignments.

`AssignPost =`

Specifies the name(s) of the assignment section(s) to be applied after an iteration of the command section has completed either successfully or in error. These assignments are applied before the next iteration of the section.l. They are applied before any AssignTerm assignments.

Load Statements

Load statements cause a record to be read from the ARS server and associated with the Tag given. They may be coded in the control section or in an assignment section.

A load statements specifies a query to be performed that will return exactly one record to load and associates that record with the specified Tag.

A **`Load=`** statement consists of the keyword `LoadQ=` and a three part value plus an optional read server reference.

```
LoadQ        =      [ @ SrvTag ]  Tag,  Schema,  Qry
```

- The `SvrTag`, if coded, indicates that this record will be loaded from the server specified as a Read Server with the matching Tag.
- The `Tag` is used as references to the loaded record's fields in assignments to the target record, in other Loads, in Queries, Launches and Updates.
- The `Schema` is the ARS form on the server to read from.
- The `Qry` is a query string whose result must return one and only one record.

Any field in the loaded record's form can be assigned to any field in the update assignments. Meta-Update does automatic type conversions. A loaded record's field can also be used as a Key in a subsequent load or inside a query string.

Two loads with the same Tag is an error.

Loads are processed in the order coded. This order may be important as a field from a loaded record may be used to load another record.

All loads specified in the control section before the `Query=`, `File=`, and `Update=` statements of that section are processed before the `Query=`, `File=`, and `Update=` statements. The `Query=`, `File=`, and `Update=` statements can use data loaded in the preceding loads.

Load statements specified after `Query=`, `File=`, and `Update=` statements are processed after these statements and can use the data in the results of the `Query=`, `File=`, and `Update=` statements.

There is no distinction between loads in a control section and loads in an assignment section other than the fact that the loads from the control section are processed first. The query and update are then processed resulting in one or two more loads. Then the assignments may use all loaded data from either section.

The **LoadQ** statement has been superseded by the more powerful LookUp facility.

Note that the LookUp facility can also be used to Load records and has advantages over the Load including caching the records, using the first record when multiple records are returned, and allowing no matching records.

If a Load query returns zero records, an error is thrown.

Query Statements

A Query statement is used to iterate through a query result of records. For each record, other records may be created or updated, and other control sections may launched and assignment sections may be processed.

All results from a query are processed even if the server limits the number of records returned. The starting record returned by the results and the maximum number of records returned by the results can be controlled if desired.

There are two types of Query statements: `Query=` and `QuerySql=`. Both types use the ARS API to return results.

A single `Query=` or **`QuerySql=`** statement may be coded in a control section.

When the `Query=` or `QuerySql=` statement is coded, Meta-Update will issue the supplied query and for each record returned in that query will:

- Load that record and associate that data with the tag specified on the Query statement.
- Perform any AssignPre section if coded.
  This is a great place to load related records, transform values, validate the record loaded, and, set the target schema for the Update.
- Perform an **`Update=`** query if coded, and, apply the assignment sections to create or update a record in the target schema.
- **`Launch`** other control sections to update other records, possibly using variable set in the **`AssignPre=`** to add a condition to the launches.

An **`Update=`** can result in the same number of new records added to, or updated in, the target schema as was returned by the query.

Syntax

```
Query         =      [ @ SrvTag ]                    &
                     Tag,                            &
                     Schema,                         &
                     [@sort(Fld[,..]),               &
                     Qualification
```

@SvrTag	Specifies a **ReadServer** to run the Query on. The **ReadServer**'s **Tag=** value is the **SvrTag** and is prefixed with an "@". The **ReadServer**'s section must be specified in the **Main ReadServer=** keyword.
Tag	Specifies the **Tag** that each of the returned records will be assigned to and referenced with. The **Tag** is used throughout the script to access data from the query result. The **Tag** is reloaded while iterating through the query results You can then use **$Tag, field$** to reference data from the record.
Schema	The Schema is a full ARS table name that will be queried. It may be a reference. An **AssignPre** section, for example, could determine a table to update and set the name of the updating schema in a script variable.
@sort(**Fld [,..]** **)**	Specifies a specific sort order. List the fields to be used in the sort by name or Id and optionally follow a field by **-A** or **Ascending** or **-D** or **Descending** to specify the previous field's sort direction. The set of fields may include references. An **@sort($NULL$)** evaluated by expanding a reference causes no sort to be applied to the query. Note that if an **@sort** is not specified, the Remedy schema specifies a default sort and this is implicitly used.
Qualification	Specifies the Query Qualification that will be passed to Remedy. Qualifications may include script references. Generally, any qualification acceptable to the advanced query bar of the User tool is acceptable. The Qualification may include Remedy field names between single quotes. Meta-Update will replace these with field Ids when the default field label is different then the field name.

The qualification string is similar to one that you would enter when issuing a query in the advanced bar with the user tool. Any literal $'s must be doubled or escaped.

Meta-Update reference substitution on the query qualification is done. This can be in any part of the qualification including the Remedy fields between single quotes.

Field Ids, field labels, and field names may be coded between single quotes. If a field name is used, and that field name does not match the default field label in the ARS schema, the field ID is substituted before the query is sent to Remedy.

The values **"$NULL$"**, **""**, and **$NULL$** are equivalent and replaced with the **$NULL$** keyword with any quotes removed.

The query may be tested using Meta-Query.

Using **`-d:q`** or **`-v`** on the Meta-Update run will cause the complete text for all query qualifications sent to ARS to be logged.

Using **`-d:q,q`** on the run will log the query sent to Remedy, and, if the Meta-Update user is in the configured client logging group, will also log the resultant ARS Server SQL logs.

To perform substitution, use assignment references wrapped in $'s. Examples are:

```
Query        =      @SrcSrv,                                    &
                    SrcR,                                       &
                    HPD:HelpDesk,                               &
                    'Key'    = "$Arg, Key$"                 AND &
                    'Value' >  $Src, TgtValue$              AND &
                    'Non-Null'    !=     $NULL$
```

If the command argument named Key had the value "**`Key1`**" and the value of the TgtValue field in the record loaded as "Src" was "**`1`**", then the substituted query qualification would be:

```
'Key' = "Key1" AND 'Value' > 1 AND 'Non-Null' != $NULL$
```

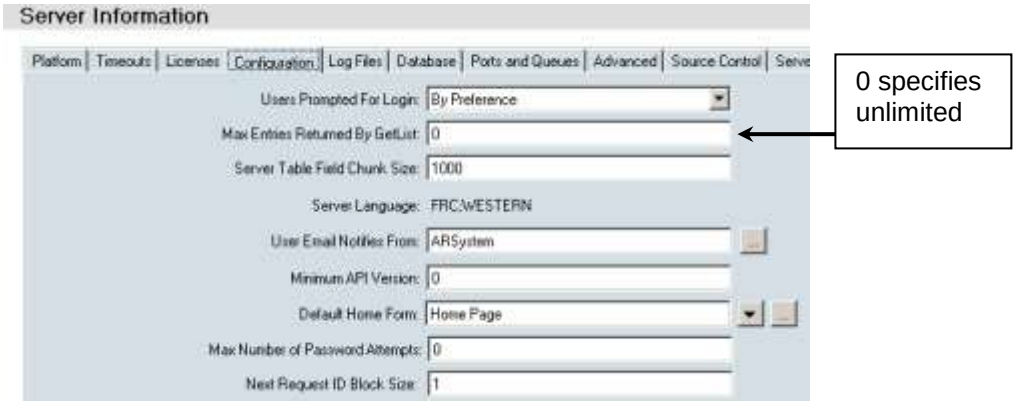
If, on the other hand, the value of Key was "" or NULL, the substituted query string would be:

```
'Key' = $NULL$ AND 'Value' > 1 AND 'Non-Null' != $NULL$
```

Note that only one of `Query=, QuerySql=, File=,` or `Loop=` may be used in any single control section.

Performance Considerations

A Query result limit may be imposed through a Remedy server configuration setting.



Meta-Update will retrieve ***all*** query results by issuing a Remedy call to get the next chunk of results until all the results are retrieved.

Once a set of Query results are retrieved, Meta-Update will retrieve the data for those results in blocks of 100 records. Each iteration of the section will load the next record from the current block until that block is exhausted. It will then retrieve the next block from Remedy.

This reduces accesses to the Remedy server to once per 100 records, and 1 per query chunk with a Remedy Server maximum, or 1 if unlimited.

Additional Keywords

The following optional keywords may be included in a control section that has a **Query=** statement.

QueryStart = nnn
QueryMax = nnn

If **QueryStart=** is coded, the first record returned by the query will be the record specified. If **QueryMax=** is coded, the total number of records returned by the query will be limited to the number given.

The values can be integers or references that evaluate to integers. If missing, the default will be the first record returned by the query.

The special integer 999,999,999 can be used to override a server-based limit on servers above release 7. It is unnecessary but possible to set this in a Meta-Update script. Meta-Update, by default, will continue issuing queries automatically until all results are exhausted.

These keywords can be used to limit the number of records processed by a single job allowing you to start multiple jobs with different **QueryStart=** values.

In this example of a script, we will run 4 simultaneous jobs with 2.5k per job:

```
[Main]
Arg         =     start
Arg         =     max
Arg         =     qry

[Do]
Query       =     Src,  HPD:Help Desk,  $Arg, qry$
QueryStart  =     $Arg, start$
QueryMax    =     $Arg, max$
```

To fire the jobs we might use a batch file like this:

```
start SthMupd -d:i,,j1.log example.ini Do -start 00000 -max 2500 -qry 1=1
start SthMupd -d:i,,j2.log example.ini Do -start 02500 -max 2500 -qry 1=1
start SthMupd -d:i,,j3.log example.ini Do -start 05000 -max 2500 -qry 1=1
start SthMupd -d:i,,j4.log example.ini Do -start 07500 -max 2500 -qry 1=1
```

QuerySql Statement

A QuerySql statement is like a Query statement except that instead of supplying a Remedy Table and Query, an SQL query is provided and that query is passed through ARS onto the database.

A single `QuerySql=` statement may be coded in a control section.

When the `QuerySql=` statement is coded, Meta-Update will issue the supplied query using the ARS API function, and for each record returned in that query will:

- Load that record and associate that data with the supplied tag.
- Perform any AssignPre section if coded.
- Perform and Update query if coded, and
- Apply the assignment sections to create or update a record in the target schema.

This will result in the same number of new records added to, or updated in, the target schema as was returned by the query.

```
QuerySql      =      [ @ SrvTag ]  Tag,  Sec,  Query
```

The statement has three parts with an optional reference to a read-server.

The **`Tag`** is the name that Meta-Update will recognise as a reference to the current record in the result set.

The **`Sec`** specifies a section that is used to specify column names and value translations as per a `Fields=` section of a `File=`. See below for more information on the `Fields=` section.

The **`Sec`** may be empty or **`@na`**. Both mean the same and there will be no column names and no data value transformations for the record denoted by the **`Tag`**. In this case, the field names are integers much like assignments with the Administrator Tool: The first column selected has the "name" 1, the second, "2" and so forth.

If **`Sec`** is not empty and refers to a field section, and automatic variable is set which holds the select fields separated by commas. This is gathered from the fields in the section. When fields are the result of complex SQL expressions, those expressions may be coded in the field section with **`Sql=`**.

This variable is **`$CTL, Sec-SqlSelect$`**

The query string is similar to one that you would enter when issuing a query in a set fields action dialogue with the BMC Administrator Tool except, of-course, that the query may return multiple records and these will be iterated through in the control section.

The query may be tested the Meta-Query.

Text substitution in the query qualification is done. Wrap references in dollar signs. Literal $'s must be doubled or escaped.

If a dereferenced Query string contains equal and not equal comparisons with '$NULL$', that comparison will be replaced with "is null" or "is not null" respectively. This qualification example should help clarify:

```
DbField = '$Tag, Ars-Field$' or DbField <> '$Tag, Ars-Field$'
```

If the value of Ars-Field in the record referenced by Tag is $NULL$, the qualification can become.

```
DbField = '$NULL$' or DbField <> '$NULL$'
```

This of course, would not match what is wanted, so the above QuerySql qualifications will be changed to:

```
DbField is null or DbField is not null
```

Examples:

```
QuerySql      =      SqlRec,  SqlFlds,                              &
                        select distinct                             &
                           Category, Type, Item                     &
                        from BMC_BMC_AssetBase

[SqlFlds]
Category      =      $
Type          =      $
Item          =      $
```

Note that the following query is entirely equivalent to the above.

```
QuerySql      =      SqlRec,  SqlFlds,                              &
                        select distinct                             &
                           $CTL, SqlFlds-SqlSelect$                 &
                        from BMC_BMC_AssetBase
```

References to **`SqlRec`** could then be coded as:

```
Query         =        ShrCat,  SHR:Categorization,                  &
                          'Category'  = "$SqlRec, Category$"   AND &
                          'Type'   = "$SqlRec, Type$"          AND &
                          'Item'   = "$SqlRec, Item$"
```

Or in assignments as:

```
Category      =      SqlRec, 1
Type          =      SqlRec, Type
Item          =      SqlRec, 3
```

Another example:

```
QuerySql      =      @SrcSvr,  SrcPct,  SrcPct,                     &
                        select Request_Id, Instance_Id,             &
                          Category, Type, Item
                        from
                          (select distinct item from $Arg, Schema$  &
                           where AssetLifecycleStatus != 5 and      &
                                 BMC_DataSet = 'BMC.ASSET'          &
                          )
[SrcPct]
RequestID     =      $
InstanceID    =      $
Category      =      $
Type          =      $
Item          =      $
```

In the above example, records of 5 “fields” or values will be retrieved. These fields are can be referenced in two ways:

1) simply by their column numbers starting from 1 as in the BMC Administrator, or,
2) by the field names in the `[SrcPct]` section.

For the `QuerySql=` above, the fields of each SQL row can be referenced as:

SrcPct, 1	SrcPct, RequestID
SrcPct, 2	SrcPct, InstanceID
SrcPct, 3	SrcPct, Category
SrcPct, 4	SrcPct, Type
SrcPct, 5	SrcPct, Item

Here are example references using the **`QuerySql=`** above:

```
Query         =      @SrcSvr,  CI,                          &
                     'Category' = "$Pct, 3" AND              &
                     'Category' = "$Pct, 3" AND              &
```

If the example QuerySql launched a section with the next example Query=, the effect would be to process all of the distinct “Items” in the Asset database, and then for each of those Items, process the set of Assets that have that categorisation.

Note: at most one and only one iteration statement: `QuerySql=`, `Query=`, `File=`, or `Loop=` may be coded in a command section.

File Statement

A single `File=` statement may be coded in the control section. Meta-Update will read the specified file record by record. For each record in the file, Meta-Update will:

- Load that record and associate that data with the supplied tag.
- Query for an update record.
- Apply the assignment sections to create or update a record in the target schema.

This will result in the same number of new records added to, or updated in, the target schema as are in the file.

```
File          =      Tag,   DefSec,       FileSpec
```

`Tag`	specifies the name that will be associated with the fields of each file’s records. This tag is used in references.
`DefSec`	specifies the file sections that define the characteristics of the ASCII file.
`FileSpec`	This is the actual file name and path. It is a string reference.

```
File  =  F_OutLook, fDefExchg, $ENV, Rmdy$work/exchg_$Arg, fname$_.cvs
File  =  F_OutLook, fDefExchg, exchg.cvs
```

The first example uses the environment variable Rmdy and the program argument **`-fname`**.

File records are similar to ARS records. A record is comprised of fields. The fields may be referenced in ARS assignments in the same way as a loaded record’s fields can be referenced.

There are two types of input files: Delimited and Fixed.

Delimited files are used by Excel. Microsoft Exchange also produces these files.

Fixed format files have fields that are always a specified length. Transaction files, UNIX script output and input files tend to be fixed format.

See File and Field section below for more information on the options you can select.

When Launching a control section with a `File=`, that File will be processed for each record in the parent control section. For example, if the first control section processes a 100 record file, and the second control section processes a 10 record file, the assignments of the second control section will be processed 100*10 or 1,000 times.

Note: at most one and only one **`QuerySql=, Query=, File=`**, or a **`Loop=`** may be coded in a command section.

Loop Statement

A single **`Loop=`** statement may be coded in the control section. Meta-Update will iterate through the values being looped. For each value in the loop, Meta-Update will set the loop reference tag and can create or update a single record.

Note: at most one and only one iteration statement (`QuerySql=, Query=, File=`, or `Loop=`) may be coded in a command section.

Types of Loops

There are four types of loops:

`String`	a string is parsed according to a specified delimiter.
`Diary`	all entries in a Diary field value are iterated through.
`While`	a loop continues while the specified condition is true.
`Fields`	all fields defined by a source Tag are looped through.
`Join`	all normal forms that make up a Join are looped through.

Syntax overview

```
Loop    =    String,   Tag,     dlm,     [ sort ]    Ref
             Diary,    Tag,              [ sort ]    Ref
             While,    ( expression )
             Fields,   Tag,     SrcTag,    [Options]
             Join,     Tag,     [ SvrTag ],  Schema
                     [ Skip: list ]
```

These are keywords that specify the type of loop and are required.

`String`	A string loop iterates through a set of substring.
`Diary`	A diary loop iterates through all diary entries in a single diary field.
`While`	A while loop iterates until the condition is false.
`Fields`	Each field of the source Tag is iterated through.
`Join`	Each normal form that makes up the Join is iterated through.

Keywords

`Ref=` This is a string reference.
If Ref evaluates to a NULL, no iterations and no outputs are performed. This is similar to a **`Query=`** that returns no records.

For diary loops, the reference must evaluate to a formatted diary field. It will generally be to a loaded record's diary field or a diary field picked up by SQL.

dlm This is a string delimiter.
A delimiter can be a single character or a string. It can also be a reference that evaluates to a single character or a string..

If the delimiter is a string, it can be "anchored" by prefixing the string with a single or double cimcumflex character "^" or "^^", or by suffixing the string with a single "$".

A prefixing "^"means that the delimiter will match only if it is preceeded by a new line or is at the start of the string. The "^" is not matched.

A prefixing "^^" means two new lines must precede the delimiter string to be considered a match.

A suffixing "**$**" means that the delimiter will match only if it suffixed by a new line or the end of the string. Newlines may be in either Windows or UNIX formats on either OS.

If the delimiter is a single character, the values iterated through the looped strings do not include the delimiter. If the delimiter is a multicharacter string, the looped strings include the delimiter string.

Delimiters may be quoted and may contain escape characters (for example "\n" or "\013").

The string is parsed into an array of strings by using the delimiter as a separator.

A non-null string with no delimiters returns a single complete string.

Sort An optional sort if specified, must be coded as one of:

```
forward   ascending   asc   reverse   decending
```

- **forward** In the natural order, that is the source order within the string or diary field. No sort is applied. This is the default setting.
- **reverse** Reverses the natural order – not the same as descending except for Diary loops.
- **ascending** Sorts set data in ascending sequence. If both are numeric uses a numeric compare else does character comparisons. May be abreviated to "**asc**".
- **descending** Sorts set data in descending sequence. If both are numeric uses a numeric compare else does character comparisons. May be abreviated to "de**sc**".

If the sort is not specified, no sort, or "**forward**" is applied.

For Diary loops the time stamp of the diary entry is used in the sort.

The ARS server stores Diary entries in an encoded diary string by appending to the string – always from oldest to newest. Hence, "**forward**" is equivalent to "**ascending**" and is from oldest to newest. "**reverse**" is equivalent to "**descending**" and is from newest to oldest.

In a String loop, the "**forward**" order is the order that the individual strings are parsed from the whole reference. Strings are compared as case sensitive strings except when both strings are integers. In that case, the integers are compared.

So:	104;17;12	will sort as expected (numerically)	
and	104a;17a;12a	will sort as	104a;12a;17a

The following string example, will illustrate the effects of the sort keywords:
String value: **1,97,42,26,51**

Forward	**1,97,42,26,51**
Reverse	**51,26,42,97,1**
Ascending	**1,26,42,51,97**
Descending	**97,52,42,26,1**

While, Field, Join loops ignore any sort.

Tag assignments

A String loop sets only a single named value into the **Tag**:

Text	The text of this iteration's string

A Diary loop sets several named values into the **Tag**:

Text	The text of this iteration's diatry entry		
User	The user value for the entry		
Date	The date the user made the entry		
Date	The date the user made the entry		"**yyyy/mm/dd hh:mm:ss**"
DateYmd	…	formatted like:	"**yyyy/mm/dd hh:mm:ss**"
DateMdy	…	formatted like:	"mm/dd/**yyyy hh:mm:ss**"
DateDmy	…	formatted like:	"dd/mm/**yyyy hh:mm:ss**"
DateI	..	Remedy timestamp value	**nnnnn**

A Fields loop sets the named values defined by the **@info** Reference assignment command (See page 183 for the complete list) into the **Tag**:

FieldName	The name of the field of this iteration
Value	The actual value of that field
ValueLength	The length of the above string

A Join loop sets the named values defined by the **@info** Reference assignment command (See page 183 for the complete list) into the Tag:

Schema	The name of one of the Normal Schemas making up the Join

Note that the field values of **@info** will not be filled in.

Examples

In the following discussion, we describe examples of a Loop statement coded in the Do-Loop section of this script:

```
   [Do]
   Query    =  SrcTT,    HPD:HelpDesk,  '1' = "$Arg,  ID$"
   Launch   =  [Do-Loop]

   [Do-Loop]
1  Loop     =  Diary,    sDiary,              $SrcTT, Notes$
2  Loop     =  String,   sTag,  ";",          $Usr,   Group List$
3  Loop     =  Fields,   fTag,  SrcTT,  Type Attachment, NoNulls
4  Loop     =  String,   sTag,  "^*** ",      $SrcTT,  CASE_HISTORY
```

Example 1 Diary:

```
   Loop     =  Diary,    sDiary,              $SrcTT, Notes$
```

In the first example, a Help Desk ticket is loaded into the Tag `SrcTT`. The Notes field, a diary field, is parsed, and each entry in that diary field is iterated through. When the entry is loaded, the following references are made available to the section:

`sDiary`	`User`	the login name of the user who made the diary entry
`sDiary`	`Date`	the date of the entry: yyyy/mm/dd hh:mm:ss
`sDiary`	`DateMdy`	the date of the entry: mm/dd/yyyy hh:mm:ss
`sDiary`	`DateDmy`	the date of the entry: dd/mm/yyyy hh:mm:ss
`sDiary`	`Text`	the entry text

The `Date` value is useful for assignments. This is the format that Meta-Update expects for date variables. The `DateXxx` values are useful for ARS Queries which require that the date be formatted according to the machine's locale. In Windows, this is set at a machine level. On Unix, the local may be controlled by environment variables. The "C" locale, a default, is referenced by `DateMdy`.

Example 2 String:

```
   Loop     =  String,   sTag,  ";",          $Usr,   Group List$
```

In the second example, a User record is loaded into the tag `Usr`. The Group List field is parsed (based on the semi-colon seperator specified) into a set of single groups. Each of those groups is interated through. When each group is loaded, the following references are made available :

sTag	Text	the single group id as a string

Example 3 Fields:

```
   Loop     =  Fields,   fTag,  SrcTT,  Type Attachment, NoNulls
```

In the third example,a Loop of only a records' attachment fields containing attachments (non-null) are iterated through.

The Tag, `fTag` will contain the information returned from the `@info` reference assignment command.

If the record has three attachment fields and two have no attachments (are $NULL$) the loop will execute once only with these fields being assigned to the `fTag` specified:

```
fTag        Type                ARS
            SchemaName          HPD:HelpDesk
            FieldName           Attachment1
            FieldId             3000100010
            FieldType           Attachment
            Value               C:\tmp\Some_File.jpg
            ValueLength         19
```

See *Assignment Reference,* on page 183, for the list of variables assigned to `fTag`.

Example 3 String:

```
Loop    =   String,   sTag,  "^**** ",      $SrcTT,  CASE_HISTORY
```

In the fourth example, say `$SrcTT, CASE_HISTORY$` contains:

```
*** CASE OPEN 2011/08/01 sup1
The customer called complaining of slow response time.  This
generally happens for a period of an hour across his lunch.

*** CASE NOTES 2011/08/02 sup1
Ran the tracert to his server when he experienced the slowness.
*** tracert output attached

*** CASE TRANSFERED 2011/08/02 sup-net

*** CASE CLOSED 2011/08/02 sup-net
Firewall change made.
```

Consider the following **`Loop=`** example using a double anchor on the delimiter:

```
Loop        =     String,                                    &
                  T,                                         &
                  "^^***",                                   &
                  $Src, CASE_HISTORY$
```

The **`Loop=`** will iterate through 4 strings. These are:

```
Lp 1 of 4: *** CASE OPEN 2011/08/01 sup1
           The customer  ...
Lp 2 of 4: *** CASE NOTES 2011/08/02 sup1
           Ran the trac  ...
Lp 3 of 4: *** CASE TRANSFERED 2011/08/02 sup-net
Lp 4 of 4: *** CASE CLOSED 2011/08/02 sup-ne
           Firewall cha  ...
```

It is up to the `AssignPre` of the Loop section to parse the looped strings and determine what to do.

When a double anchor is used, only when the delimiter string is preceded by two new lines does that string be considered a match.
Consider the following example:

```
Loop        =     String,                                    &
```

```
                                T,                                          &
                                "^^***",                                    &
                                $SrcC, CASE_HISTORY$
```

Then the loop will be executed four times. If, on the other hand, a single anchor were used, the loop would be executed five times and iterate through these strings:

```
Lp 1 of 5: *** CASE OPEN 2011/08/01 sup1
           The customer  ...
Lp 2 of 5: *** CASE NOTES 2011/08/02 sup1
           Ran the trac  ...
Lp 3 of 5: *** tracert output attached
Lp 4 of 5: *** CASE TRANSFERED 2011/08/02 sup-net
Lp 5 of 5: *** CASE CLOSED 2011/08/02 sup-ne
           Firewall cha  ...
```

Example 5 Join

The following script will transfer records from a class form on a production server to a target server. Because we do not want workflow to fire, we will use the Merge API and write the records to the underlying normal forms.

The class form as well as query qualifications are passed on the command line.

```
[Main]
Arg             =   Form,     Default BMC.CORE:BMC_ComputerSystem
Arg             =   Query     Default 1=1
ReadServer      =   prod
PrmReq          =   Function:
PrmReq          =   . Will transfer CMDB records from production.
PrmReq          =   Usage:
PrmReq          =   . SthMupd $ScriptFx$ Do –Form form –Query query
PrmReq          =   .   where form is a BMC.CORE:BMC_xxx class form
PrmReq          =   .   and   query is a qualification on the form.

[prod]
Tag             =   prod
Server          =   $ENV, ArsProdServer $
User            =   $ENV, ArsProdUser   $

[Do]
Query           =   @prod, Src, $Arg, Form$, $Arg, Query$
Launch          =   Do-Join

[Do-Join]
Loop            =   Join, SrcI, $Arg, Form$
Launch          =   Do_I

[Do_I]
Query           =   @prod, SrcIr, $SrcI, Schema$, '179' = "$Src, 179$"
Update          =   TgtIr, $SrcI, Schema$, '179' = "$Src, 179$"
Merge,          =   Yes, NoWorkflow
AssignNew       =   Do_I-asg
Assign          =   Do_I-asg

[Do-I-asg]
@Cmd            = Copy, SrcIr, CoreAssign
```

[Main] sets script arguments and sets up 2 connections: a "prod" server, and the target server.

Environment variable **ArsSvrAdmin** is the target server. **ArsSvrProdAdmin** is the "prod" server.

[Do] issues a Query on the BMC Class Join form given by the program arguments.

[Do-Join] loops through each normal form in the class join.

[Do-I] issues a query on each normal forms of the class join and updates the records on the target server.

If the above script were called as follows:

```
SthMupd.exe CmdbXfer.ini Do –Form BMC.CORE:BMC_Mainframe
```

All mainframes, no matter what datasets, would be transferred from the production server to the target server.

The following output might result:

```
[Do] Qry 1 of 1 BMC.CORE:BMC_Mainframe RE7269hqy01mna6y01qa
[Do] Qry 1 of 1: Launching Do-Join
  [Do-Join] Lp 1 of 3 BMC.CORE:BMC_Mainframe_
  [Do-Join] Lp 1 of 3 Launching Do_I
    [Do_I] Qry 1 of 1: BMC.CORE:BMC_Mainframe_ RE7269hqy01mna6y01qa
    [Do_I] Updated BMC.CORE:BMC_Mainframe_ RE7269hqy01mna6y01qa
  [Do-Join] Lp 1 of 3 BMC.CORE:BMC_ComputerSystem_
  [Do-Join] Lp 2 of 3 Launching Do_I
    [Do_I] Qry 1 of 1: BMC.CORE:BMC_ComputerSystem_ RE7269hqy01mna6y01qa
    [Do_I] Updated BMC.CORE:BMC_ ComputerSystem_ RE7269hqy01mna6y01qa
  [Do-Join] Lp 1 of 3 BMC.CORE:BMC_ BaseElement
  [Do-Join] Lp 3 of 3 Launching Do_I
    [Do_I] Qry 1 of 1: BMC.CORE:BMC_BaseElement RE7269hqy01mna6y01qa
    [Do_I] Updated BMC.CORE:BMC_ BaseElement RE7269hqy01mna6y01qa
  [Do-Join] Lp completed 3 records OK
[Do] Qry completed 1 record OK
```

Create Statement

A single **Create=** statement may be coded in the control section if Meta-Update is to always create new records with every iteration.

Note that generally it is better to code an **Update=** so that when the same script is run, the pertinent records are updated rather than created.

A **Create=** statement has only two parts. The Tag that the created record will be known under in any Launched sections, and the schema to create. After a record is created, it is reloaded into the Tag so that Launches will have available all values of that record.

All loads in the control section are processed before the `Update=` is processed. A `Query=` or `File=` is processed before the `Update=`. The assignment section is processed after the `Update=` is processed.

Until Statement

The **Until** statement may be used only when a control section includes an iteration such as **Query=**, **QuerySql=**, **File=**, or, **Loop=**.

The **Until** statement specifies a condition, that when true, causes the control section to stop its iterations with no errors.

```
Until        =      @if(condition)
```

If an error is needed, use the **Abort** assignment command.

In a section that does not iterate, any **Until=** statement is ignored with a warning.

In the following example, an infinite loop is set up but is aborted after a single iteration.

```
[Do]
AssignInit  =     Do-asgInit
Loop        =     While(1)
Until       =     @if(1)

[Do]
AssignInit  =     Do-asgInit
```

These two sections are entirely equivalent

An **Until=** statement can be used to limit a section's processing when any condition becomes known. For example, say you want to delete an Incident and all its dependencies but only if a copy of that Incident and all those dependencies exist in an archive form.

Consider this example, where we want to validate that a root request and all its children exist in alternate – or archive – forms. If any child is missing, there is no point continuing. This request has failed the validation.

So, say you may have sections querying all of the Incident's children on the real forms, and a LookUps to check the Archive forms. When the first case of a missing child is found, there is no point continuing any of the queries for this incident's children, so a flag can be set and all Launched sections would complete up to but not including the section that looped through the Incidents. That section would then iterate to the next Incident.

In this example, the flag **V, Do**, is initialized to True and set false when a Work Log for this incident is not found in the archive form.

```
[Do]
Query       =  Inc,                                  &
   HPD:Help Desk,                              &
   qry
AssignPre   =  Do-asgPre
Launch          =  @if("$V, Do$")  Do-WL
Launch          =  @if("$V, Do$")  Do-delete

[Do-asgPre]
@Cmd        =  Ref, V,   Do,   1
@Cmd        =  Ref, V,   gotIncArch,   @LookUp,        &
         Lkp-Inc-Arch,        $Inc, 179$
@Cmd        =  @if(! "$V, gotIncArch$")          &
         Ref, V,   Do,   0

[Do-WL]
Query       =  WL,                                   &
         HPD:WorkLog,                          &
        'Incident Number' = "$Inc,  Incident Number$"
Until       =  (! "$V, Do$")
AssignPre   =  Do-WL-asgPre

[Do-WL-asgPre]
@Cmd        =  Ref, V,   gotIncWL,     @LookUp,        &
         Lkp-Inc-WL,          $WL, 179$
@Cmd        =  @if(! "$V, gotIncWL$")                &
         Ref, V,   Do,   0
```

The **Until=** stops the processing of this Incident's Work Logs as soon as a Work Log record is discovered missing from an Archive form.

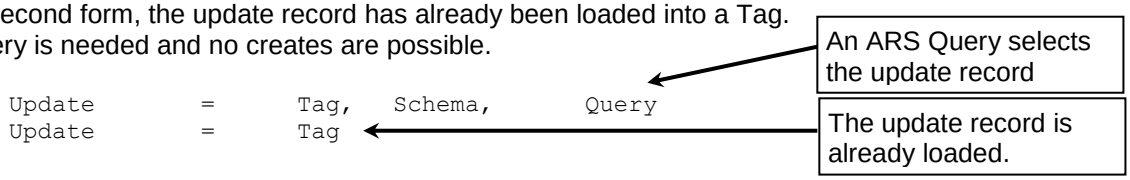
If a Work Log is missing from an Archive form, we set the **V, Do** flag to false.

Update Statement

A single `Update=` statement may be coded in the control section if Meta-Update is to update existing records in the target form, or create records if the specified `Update=` query returns no records. .

There are two forms of the Update= statement. In the first form, a query is performed to determine the update record, or determine that a new record needs to be created.

In the second form, the update record has already been loaded into a Tag. No Query is needed and no creates are possible.




```
Update        =      Tag,   Schema,       Query
Update        =      Tag
```

If Meta-Update is to always create new records without issuing any query, use `Create=` instead of `Update=`.

All loads in the control section are processed before the `Update=` is processed. Any iteration statement (`Query=`, `QuerySql=`, `File=`, or, `Loop=)` is processed before the `Update=`.

The assignments sections which will set the fields of the **`Update=`** record, are specified by **`Assign=`** and **`AssignNew=`** keywords. When all assignments are done, the record is updated on the server.

The **`Update=`** statement issues the ARS query to load the Tag. If the query returns no records, no Tag is loaded. If and only if there is a list of assignment sections specified in the **`AssignNew=`** keyword, these are taken and a new record is created. If there are no **`AssignNew=`** sections, no new records will be created. No error is thrown.

The **`Update=`** Tag is automatically loaded with a new copy of the update record after the update is done.

This cannot be done on Join forms. A Warning is issued if the **`Update=`** form is a join form and the Update Tag will be undefined.

You can use an **`AssignPost=`** section to reload the Tag yourself in the case of Join forms.

The Schema is a form name and may be a reference.

The Query is any valid Remedy Query.

```
Update        =      Tag,   Schema,       Query
```

If a section wants to update a Tag that is already loaded, as a result of a Query=, a LookUp, or a **`LoadQ=`**, then the Tag may be specified alone in an Update statement:

```
Update        =      Tag
```

The Tag must have been previously loaded and must be a Remedy record.

Consider this example:

```
[Do]
Query          =  Hpd,       HPD:Help Desk,       Query
AssignPre      =  Do-asgPre
Launch         =  @if("$V, gotPplAsg$")   Do-Asgee

[Do-asgPre]
@Cmd           =  Ref, V,    gotPplAsg,           &      &
                  @LookUp,  Lkp-Ppl-Asg, $Hpd, AssigneeId$

[Do-Asgee]
Update         =  PplAsg

[Lkp-Ppl-Asg]
# given a person's ID, load CTM:People rec into PplAsg
Cache          =  100
Default        =  0
NoMatch        =  D, Default
Query          =  PplAsg,    CTM:People,                &
                  '1' = "$CTL, LookUpSrc$"
QueryTarget    =  $PplAsg, 1$
```

Loads **PplAsg** with a record from **CTM:People**

We update the Loaded **CTM:People** record held in **PplAsg**, the update record. **PplAsg** data is replaced with the re-read, updated record

In this example, we want to update the records returned by a query:

```
[Do]
Query          =  Hpd,       HPD:Help Desk,       Query
Update         =  Hpd
```

The `Update=` query results must include exactly one record. That record is updated and loaded. It is an error for the `Update=` query to return more than one record.

If the `Update=` query returns no matching records, you can instruct Meta-Update to create a new record with the `AssignNew=` statement. This overrides the default behaviour of returning an error.

The `AssignNew=` specifies a list of assignment sections to be applied to the new create. On create, you may want to assign values to more fields. You can specify the same sections as in the Assign= or a different set of sections.

```
Assign         =      PplAsg
AssignNew      =      PplAsg, PplDft
```

In the above example used to load SHR:People from an Exchange Post Office extract, normal updates use the assignment section `[PplAsg]`. New submits include additional assignments contained in section `[PplAsg]`.

When an **Update=** is processed against a record, and that record already exists, the fields assigned are compared to their values in the current record. If there are no changes, by default the update is skipped but counted as successful.

In some cases, this may not be desired. A special keyword can be used to override this behaviour.

The **UpdateIfEqual = Yes** statement is coded in the same section as **Update=** will force the Update= to always write the Update to Remedy.

```
[Do]
Update        =  Hpd
UpdateIfEqual =  Yes
Assign        =  asg

[asg]
Short Description   = Hpd, Short Description
```

In the above fragment, the Short Description field is assigned the same value as it already has. By default this will skip the actual Remedy update. Because the `UpdateIfEqual= Yes` statement is in the Update= section, the real update to Remedy will fire. This can be used to force workflow firing for example.

Output Statement

A single **`Output=`** statement may be coded in the control section if Meta-Update is to output either a CSV row, more text to a pattern file, or a completely new pattern file.

All loads in the control section are processed before the `Output=` is processed.

An **`Output=`** can be part of the iteration that is the result of a **`Query=`** or **`File=`** or other iteration statement.

Assignment sections are processed in each iteration and result in a new row being appended to the output file, for delimited files such as CSV, or more text being appended to a pattern file, or that text becoming a new file in a pattern file with the **`MultiFile`** option..

```
Output        =      Tag,   File-Section,        FileName
```

The **`FileName`** argument is a string expression and is evaluated once when the **`Output=`** is first encountered at the section initialization.

If the file type is **`Pattern`**, **`MultiFile`**, the **`FileName`** is re-evaluated each time that output is processed and a new file, rather than a new record, is created.

It is possible to open multiple CSV files with a single output statement. When a section is Launched with an **`Output=`** statement, the file name and tag is evaluated.

If both are unique, a new file is opened. If a tag was used before, the file name must match the name opened when that tag was used before. If so, a new row is appended to the file. If not, an error is thrown.

The sample script, Tbl-All-Bkp.ini uses this feature to produce different CSV files for a set of different tables using the same **Output=** statement:

```
[Do]
QuerySql      =      Tbl,  SqlFlds,                          &
                     select  name, viewname , schemaid &
                     from    arschema                        &
                     where   $Arg, sch-qry$
Launch        =      Do2

[Do2]
Query         =      Src,  $Tbl, name$,  1 = 1
Output        =      $Tbl, viewname$,    Out-f,          &
                     $Arg,  F-out$-$Tbl, viewname$.csv
Assign        =      asg

[asg]
@Cmd          =      Copy, Src

[Out-f]
# This declares the output CSV file.  All fields
#   from the schema are copied to the CSV
Type          =      Delimited, ",", FldHdr
Format        =      Quoted always Quotes escape lf escape
Fields        =      Out-f-flds

[Out-f-flds]
@Cmd          =      Copy, $Tbl, 1$

[Sql-Fields]
# used to name the SQL fields (rather than 1, 2, 3)
name          =      $
viewname      =      $
schemaid      =      $
```

A list of tables are returned from the first **QuerySql=** on **arschema**

For each table, a Query is run returning all records and a new output CSV is created. That file contains a column for each field in the table.

Let's say the **sch-qry** argument is "**name like 'HPD:%'**"

The second section [**Do2**] is launched once for each table returned from the SQL query against arschema.

Let's say, one of the tables returned is "**HPD:Help Desk**". When [**Do2**] is launched, a query returning all records will be run against **HPD:Help Desk** and each record will be copied to an output file.

The Tag for the **Output=** will resolve to **HPD_Help_Desk** and the file name will be suffixed by **HPD_Help_Desk.csv**. As this will be the first time this combination is encountered, a new CSV file will be created and it will contain all the fields in the Incident schema.

Now, ;et's say, one of the tables returned is "**HPD:Worklog**". When [**Do2**] is launched, a query returning all records will be run against **HPD:Worklog** and each record will be copied to an output file.

The Tag for the **Output=** will resolve to **HPD_Worklog** and the file name will be suffixed by **HPD_Worklog.csv**. As this will be the first time this combination is encountered, a new CSV file will be created and it will contain all the fields in the Incident's Worklog schema.

Merge Statement

```
Merge       =       Off
Merge       =       Yes
Merge               [No]AllowNull , [No]SkipPatternMatch
                    [No]Workflow
```

The value supplied to the **Merge=** may be a reference.

By default, that is, without a **Merge=** statement, or with a **Merge = Off** statement, Meta-Update uses a Submit or Modify API operation. Workflow set on Submit and Modify fire.

You can tell Meta-Update to use Merge similar to the way the ARImport tool operates. Only workflow that fires on Merge will be executed (by default).

Using Merge= with any value but **Off** tells Meta-Update that you want to use Merge.

You can also use a **Merge=** option to inhibit all filters including those set to fire on Merge. Use this option with caution. For example, when the output is to a join form, only workflow set to fire on Merge will allow the real underlying records to be updated. A write to a join form without underlying workflow causes no database update.

When a record is being updated, or when a field id is explicitly included in the update, the setting corresponding to “Update Old Record with New Records Data” is always used.

Status Statement

A single Status= statement may be coded in the control section. Its function is simply to issue status messages while processing a File= or Query=. These are informational messages sent to the log file and copied to stderr.

```
Status      =       1, $RecCtr$: $Rec:72$
```

The above is the default Status= specification used if none is coded. The status message is repeated every record. The text of the associated message comprises the current record number (from either the query or file) and the first 72 bytes of the record or query result string.

Any values permitted in Query= statements may be used. For example, when operating on a query based on a Help Desk schema, you may code:

```
Status      =       1, $RecCtr$: Tkt: $HD, Ticket ID$ - $HD, Summary$
```

The format of the Status message is also used to report the original record being worked on if any errors occur in the processing of that record.

You may inhibit the status message as well as the end of iteration status message with:

```
Status      =       0
```

Sleep Statement

A single `Sleep=` statement may be coded in the control section. Its function is to act as a governor for Meta-Update. You can use it to reduce the load that Meta-Update will place on the ARS server.

```
Sleep       =      recs, secs
```

The above `Sleep=` will cause Meta-Update to pause `recs` seconds every `secs` records while processing a `File=` or `Query=`. If `Sleep=` is not coded, there is no pausing.

Launch Statement

Meta Update allows you to follow chains of linked records. One control section can launch other control sections, which can, in turn, launch still others.

For example, let's say you have the following tables

Organisation 1:many Sites 1:many Services

You want to write a script to invalidate all Services belonging to an Organisation.

You write a Meta-Update control section that queries for the single Organisation record you wish to invalidate services for. This control section launches a second control section that queries for all sites associated with this Organisation.

That second control section processes a set of Site records and for each of those Site records, launches a third control section that queries for all services associated with that single Site record being processed.

That third control section invalidates the Services records for each Site of the Organisation.

```
[Org]
Query   =  Org, Organisation, '1' = $001
Launch  =  Site

[Site]
Query   =  Site, Site, 'Organisation ID' = "$Org, 1$"
Launch  =  Services

[Services]
Query   =  Service, Services, 'Site ID' = "$Site, 1$"
Assign  =  ServiceInvalidate


[ServiceInvalidate]
Status  =  Inactive
```

Launches can be made conditional. That is, the Section name launched can be build and selected from ARS data or other loaded data.

```
Launch  =  @if("$Cfg, DoHtml$") Do-Html
```

IdLog Statement

An IdLog is used to create a delimited file with a row for each record processed (read, queried, updated). You can specify multiple IdLogs and events that the IdLog will be created for. You can also control the format and content of the IdLog fields.

An IdLog is primarily used to track errors so that a subsequent Meta-Update run can process only those records that resulted in errors.

Syntax and usage of IdLogs:

```
IdLog  =      Tag                                    &
              On          Event1[,Event2...]         &
              Fdef        def_section                &
              Fname       file_name                  &
              Key         key                        &
              Fasg        assign_section
```

`Tag` The Tag identifies an IdLog file and file section. If the same Tag is used in two different IdLog statements, they must specify the same file and file definition section or an error will result. You may change assignments and events on different IdLog statements in Launched sections

A special Tag is used to turn off all Id logging for a section:

```
IdLog  =      Off
```

`On` Specifies a series of events for which this Id Log will be written to.
Events are keywords and must be coded exactly as follows:

Update	a record is updated (successfully or not)
UpdateErr	a record update failed
Create	a record is created
CreateErr	a record create failed
Iter	an iteration is done: the next record, SQL row, or file row is read, or the next loop value is processed
IterErr	an iteration failed.

`Fdef` Optional. Specifies a file section.
The IdLog will be written with the fields and attributes of this file section.

Automatic fields, if not specified, will be added in sequence after the last field specified. To change the order of the automatic fields in the output file, specify them in the file section.

If a file definition is not specified, only the automatic fields will be in the IdLog file.

These are:

Time	The time of the event.
Server	The ARS server name
User	The ARS user
Schema	The schema or file name. NULL for SQL queries.
Key	The value of '1' for ARS schemas, of a string made of the first few attributes for SQL queries, files, and Loops.
Operation	One of: Read, Update, Create
Op2	Either blank or "Merge".
Result	One of: OK, Err, or Err followed by an error message.

`Fname` Required when any IdLog Tag is first used. Optional otherwise. It is an error to use a Tag twice with different Fname values.
Specifies the name of the output IdLog file associated with this Tag.

`Key` Optional. A reference string that is used instead of the default key value generally containing references from the sections iteration Tag.

`Fasg` Optional. A single assignment section name.

This is used to override the default assignments as given above or to assign values to other fields defined in the file section for this IdLog. For example you could add some fields from an SQL query to the IdLog and use the assignment section to assign values to the added fields and even to change the assignment to automatic fields such as the Schema or Key field.

Only a single assignment section can be specified. That assignment section can include other assignment sections, record loads, lookups, spawning of server or client processes and so on.

As many IdLog statements as needed may be coded in control sections.

When a section with an IdLog launches other sections, the IdLogs are carried through to that launched section unless that launched section has its own IdLog statements.

IdLogs are recognized as the same when they have the same Tag. It is an error to specify two IdLog statements with the same Tag and different file names or different file sections.

You can use the same Tag in a Launched section's IdLog to specify a different assignment section. That section could include the launched section's IdLog assignments if needed.

Once an IdLog event is taken, and before the assignment section is started, some Tags and fields are assigned values. These can be referenced in the IdLog assignment section.

CTL	**IdLogging**	1 or 0 to indicate that IdLogging is turned on
CTL	**EvSec**	the section name of the last Id Log event

The Tag for the next variables is a concatenation of "**CTL-**" and the section name – as given by **$CTL, EvSec$**.

CTL-EvSec	**EvIdLog**	the Tag from the IdLog statement for this event.
CTL-EvSec	**EvName**	the Event name (one of the Event keywords that can be specified on the **IdLog=** statement as described above.

CTL-EvSec	**EvSch**	the Schema name which may be “” when there is no schema.
CTL-EvSec	**EvServer**	the ARS Server of the event.
CTL-EvSec	**EvUser**	the ARS Server’s User of the event.
CTL-EvSec	**EvRc**	the event’s error code (0 means no error).
CTL-EvSec	**EvOp**	the event operation (see below).
CTL-EvSec	**EvOp2**	either “Merge” or “”
CTL-EvSec	**EvKey**	the “Key” value as specified in the IdLog statement or the default key value for the type of event.

These references may be used in an event’s assignments.

File Sections

A file section defines an external ASCII file's format.

> **File=** keyword iterates through a CSV.
>
> **Output=** adds a new CSV record or appends text to a single or new file.

Files sections can be used for input or output with the **File=** and **Output=** command section keywords.

Input files are always columnar (CSV like) and can be of two types: Those with fixed length fields and those with variable length fields.

Output files may be columnar like input files or they may be pattern files. A pattern file is any type of **asci** file and can be used to generate **html** or **xml**.

A file definition can also include field definitions. These fields can specify value transformation and interpretation rules.

If an output file is columnar, fixed or delimited, the fields must be defined.

Input files have their field definitions as optional if the field includes a field header. However, it is recommended that the fields be defined in the script as well both to validate the file, and to perform value interpretations and field requirement checks.

An output file can also be a "pattern file" or a non-columnar, text file. Pattern files are generally used for reports, emails, XML, HTML, and so on. They are ASCII files.

A pattern file can be used to append text for each iteration to a single text file or to create a new output file each iteration and have the contents of that record (and of course all variables which could have been a loop of records) used in the creation and naming of the new file.

Fixed format files have fields defined in the script that are always a specified length. Transaction files, UNIX script output and input files tend to be fixed format. Fields can overlap.

Excel generated CSV files are the most common of delimited files.

Delimited files have columns that are variable in length and may be null. They have a separator character between the column values. They can have quotes around the values and, if quoted, the values can contain embedded carriage returns and line feeds. Fields cannot overlap.

For Output files, field values with embedded line feeds may cause some tools such as Excel or the BMC Remedy Import Tool to interpret the line feed as a record end. Fields with line feeds can be changed through using the line feed options in field formats.

The first record of a delimited field can contain the field names – as in most Excel generated CSV files.

For an input file, if a field header row is specified, and fields are specified in the File Section of the script that defines the file, then only those fields that are in the script file are defined and available to the script. If a script field is not in the source CSV, that field will always have the $NULL$ value. Extra fields in the source CSV are ignored.

For an output file, `FldHdr` causes the field row to be produced on open. The field order is as specified in the field section.

```
[File Def]
Type      =      Delimited, "\t"  [ , FldHdr ]
Type      =      Fixed
Type      =      Pattern  [, MultiFile ]

Field     =      Field Def
Format    =      [ Csv | Excel ] [[,] Overridden | Merge ] [format]
Quote     =      c  [ , [No]LineSpan ]
Trim      =      xxx   [ , trailing ]  [ , leading ]


[Field Def]
Field     =  1      [ Formatting ] [ # comment ]   for Delimited files.
Field     =  1, 10  [ Formatting ] [ # comment ]   for Fixed files.
```

File Section Keywords	
Type	**Delimited, ",", [, FldHdr]** Required. Default: none **Fixed** **Pattern [, Multi]** **Output= only** Specifies the type of file that will be read or written. **Fixed** format files have fields but no field header in the file and each field has a starting and ending column number. Fields may overlap. **Delimited** files are typically comma separated values (CSVs) that many software products export and import. The delimiter may be specified as a reference. The first row of the file may be the names of the fields and is specified by the keyword **FldHdr** Pattern files are used for output only and are plain ASCII text with no fields defined. The keyword Multi indicates that a new file is created on each output.
Field	**section** Default: none Specifies a section name that describes the fields of the file. Required for output files and for both output and input fixed files. Optional for input Delimited files. Not permitted for pattern files.
Format	**[Csv	Excel] [, [Overriden	Merged]]** **format spec** Default: see below Specifies a default format spec for all fields of the file. Overridden or Merged indicates what should be done with a field format spec. The default is Merged, that is both the file and field format spec will be merged to interpret the field.
ErrorFile	**filename** Default: none Specifies the name of a file (may be a reference) that will contain an exact copy of the input file records that caused an error in the script. Any error in any launched section will result in the record being added. Formatting is not applied to records in this file.

FieldsRequired	**all | field, . . .** Default: none Specifies a set of fields that cannot be NULL on either input or output. When a record is read that has a NULL in one of these fields, an error is thrown, and the record is placed in the ErrorFile if one was specified. The script does not get a chance to work with this error record.
FieldsInFile	**all | field, . . .** Default: none Specifies a set of fields that must be present in the file. Normally, if a field is declared in the script, and the actual file does not include that field, all values for that field return NULL. With this keyword, such a file will be in error. Only appropriate for input files.

`Type=` Required.

`Type=` Delimited.
For Delimited files, the second parameter specified the set of characters that can be used as the delimiters. These should not occur in the data values unless the values are quoted. The delimiter itself can be specified as a reference.

The optional `FldHdr` keyword indicates that first row of the file contains the names of all the fields.

`Type=` `Pattern, MultiFile`
For Pattern files, the optional, **`MultiFile`** keyword may be appended. This causes every `Output=` to generate a new file. The name is dereferenced each time the file is to be opened. It should dereference into a different file name.

`Field=` Optional for Delimited; Required for Fixed and Output files. Not used on patter files.

This specifies the field section for the file. That field section defines the names and positions for each column of the file.

For Delimited files, not having the `Field=` statement implies `FldHdr`. That is, if there is no field definition for a file, the first row of the file must contain the fields.

If both `Field=` and `FldHdr` are coded for input files, the fields of the file are ordered as they are in the file. Missing fields have the value $NULL$. Extra fields are ignored.

`Format=` Optional.
This specifies a default format for all fields in the file.
The format tells Meta-Update how to interpret the value of the file's field on input and how to format the field value for output.

Format specifications are described below in the Field Section.

The special keywords, Csv or Excel are equivalent for input files and stipulate a specific default format as follows:

```
Quote \" Quoted asneeded Quotes double Nulls std lf unix
```

The keyword Overridden or Merged tell Meta-Update that the field level format either overrides or is merged with the file level format. The default is Merged.

`ErrorFile=` Optional for any type of input file. Ignored on Output files.

Gives the name of an output file to build. This file will contain all those input records that resulted in an error. The supplied name is a string reference. The file so created will be in the same format as the input file. If the input file had field headers, the output file will also have field headers. No manipulation of the record data is done before output when any error is diagnosed within the processing of an input record.

The following keywords have been superseded by the Format= keyword. Their use is not recommended.

`Quote=` Optional for Delimited; ignored for Fixed.
Specifies that the fields in the file *may* be quoted. Specifies the single quote character (not escaped).
`LineSpan` is used to indicate that a quoted value within the file may contain embedded line feeds or carriage returns. The default is `NoLineSpan`.

`Trim=` Optional for any type of file.
Indicates how fields containing leading and trailing spaces are to be handled. As many trim statements as required may be coded.
The field name "`All`" is a shortcut for every field of the file. If `Trim = All is` coded, any other `Trim=` are ignored.
The default for a `Trim=` is `trailing`. You must specify both `leading` and `trailing` if you need both.
With a `trim=`, a field containing all spaces is equivalent to a zero-length field or $NULL$.

`Trim = All, leading`		Only leading spaces are trimmed.
`Trim = All, trailing, leading`		Both leading and trailing spaces trimmed.
`Trim = All, trailing`		Only trailing spaces are trimmed.
`Trim = All`		Equivalent to `All, trailing`

Fixed files contain fields that start in the same column of each record and are of a fixed length. These files must have their fields specified in a field definition section. Each field has a starting column number and a length. It is possible to have overlapping field definitions in a fixed file.

Software Tool House Inc

Field Sections

About Fields and Formats

What is a Field Section?

Field sections are a list of field, giving them names and optionally, formatting and interpretation rules. These fields are then used in those Tags for assignments.

```
[FieldSection]
ID              =       $               [format]        # a delimited field
UpdateText      =       $               [format]        # one that could span lines
```

The above example defines two fields called ID and UpdateText. Note that the required dollar sign is a simple place holder for the next position

There are two types of field sections based on the possible length of the fields: fixed or variable length.

Most columnar files, such as CSVs, are variable length. That is, each field value will be as long as it needs to be to hold the value. This includes the length zero, which Meta-Update translates (by default) to $NULL$.

Fixed fields are rarely used now. These are used by files with **`Type`** = **`Fixed`**. Fixed field begin in a column and are a specific length. Fields can never contain no value. A sequence of spaces can be used to mean $NULL$. Fixed fields can overlap on read files offering different ways to interpret the same data.

```
[FieldSection]
ID              =       1,  15          [format]
ID-Prefix       =       1,   3          [format]
ID-Suffix       =       3,  12          [format]
UpdateText      =       2, 512          [format]
```

Assignments to overlapping fields of output fixed length files are done in the order they are encountered. Such overlapping assignments would not normally be done.

Reading overlapping fields works as expected with each field having an appropriate, interpreted, value.

Where are Field Sections used?

Field sections are used with

- **`Fields=`** keyword in file definition sections
- **`QuerySql=`** field sections in command sections or look up sections
- The Assignment command: **`Ref ... @regex`** extract field sections
- The **`IdLog=`** statement to change the default fields of an IdLog output file

Only **`File=`** fields have a file or file name associated with them. Only **`File=`** fields can be of the **`Type`** = **`Fixed`**, that is with a starting and ending column number.

Specifying field order

The field sections positions are specified in different ways for fixed length fields and variable length fields.

Only fixed length fields need both a starting column and a length. Variable length fields simple have a "$" to represent the next sequential position of the field.

For variable length fields the field position is that of it in the list in the field section in all cases, except in the case of an input file ***with*** a field header row.

For input files with field headers, as specified by **`Type = Delimited, ",", FldHdr`** in the file section, the field order is given by the file's field header row.

Fields in the file and not in the field section are ignored. Fields in the field section but not in the file are assigned $NULL$ (by default).

If an input file does not contain a field specified in the field section, the value of a reference to that field will always be $NULL$.

A **`FieldsRequired=`** or **`FieldsInFile=`** keyword may be specified in the file section to throw errors as needed.

About Field Formats

Field Formats allow character substitutions and value interpretation rules to be specified when loading the field with

- a value from a file by reading the next row of the file,
- an SQL result column,
- a Regex extract

They also applied when a field is output to a file row including the **`IdLog`**.

Formats may be used to:

- interpret dates
- substitute or remove characters
- handle line feeds
- handle embedded quotes

Copying Fields from Schemas

Fields may also be copied from ARS Schemas. A simple "Copy" command can specify an ARS server and form whose fields will be included in the file.

```
@Cmd          =      Copy, [ @ ReadServer , ]
                     Schema
                     [, NoDisplayOnly ]
                     [, Skip: field [, field ... ] ]
```

The ReadServer is optional and refers to a ReadServer Tag.

The Schema may be a constant or a string expression that evaluates to a Schema name.

If the keyword `NoDiplayOnly` is specified, Display Only fields are not included as part of this file. If this is missing, all fields are added to the file's field section.

Skip allows you to specify a list of field names or field ids to not include in file'

Examples:

```
[FieldSection]
ExtraField          =      $      [format]
@Cmd                =      Copy, @ITSM6SVR, HPD:HelpDesk
AnotherExtraField   =      $
```

Or

```
[FieldSection]
ExtraField          =      $      [format]
@Cmd                =      Copy, @ITSM6SVR, $MyVars, Schema$
AnotherExtraField   =      $
```

Field Formats

Any field can have special interpretations applied either in addition to the interpretations applied to all fields in the file or overriding those applied to the whole file. Note that field sections used referenced by non-files have no global formats applied.

The format is a string following the column or width in a field specification. That string is made up of a set of keywords and values coded in any order.

Keyword	**Values**	**Meaning**
Trim [quoted]	leading trailing both	Specifies that leading and or trailing white space in the source field is to be deleted. If the field value is quoted, trimming does ***not*** take place unless the special keyword quoted is specified.
Case	upper lower title	Specifies that alphabetic characters are to change case to the specified case. For title case, the first letter or a word beginning the string or following white space is converted to uppercase and all other alphabetic characters are converted to lower case. If a word does not begin with an alphabetic character, no characters of that word are converted to uppercase. “aAbB 1Aa b1B” becomes “Aabb 1aa B1b”
Date	Spec; “Spec”	Indicates that the field contains a date value and determines how to interpret the date value. Described further below.
Quoted	never always asneeded	Indicates that the field may be surrounded by quotes and that the quotes are not part of the field value. Unless specified, the quote character is the double-quote (**“**).
Quote	std \nnn “ ‘	Indicates the single character that is to be interpreted as a quote. “Std” means the standard double-quote character. \nnn defines an ASCII character with the value nnn. “x” represents any single character.

Quotes	double escape delete	Indicates that values containing the quote character have that character either quoted or escaped. Delete is not applicable for input files.
Lf	unix nt escape delete	Quoted values can span lines. All line feeds are converted to single <lf>s in internal (ARS) values. For external values (such as output files), if this format is not specified, the default will be unix or nt based on the platform that the job is being run on.
Nulls	Std $NULL$ “” “xxx”	Specifies the conversion for Null (empty) values in the file. “std” indicates that the Remedy keyword $NULL$ will be substituted. “” Indicates that an empty string will be substituted. Note that in Remedy, as assignment of $NULL$ is not equivalent to an assignment of an empty string.
Subst	/[^]yyy[$]/xxx/	Specifies simple character based value substitutions Multiple “Subst” keywords may be coded. They are effected in the order coded. The slash (/) in the above example is a separator character. It can be any character not in either the pattern or substitution strings. In the pattern string, a leading circumflex (^) indicates that the value must begin with the pattern to be considered a match. Similarly, a dollar as the last character of the pattern says that the value must match the pattern at the end of the string. If both are specified, the value must match completely. The ^ and $ character must be escaped if they are part of and begin or end the pattern string.
Trunc	nnn	Truncate the value. Applied last.
WordChars	delete escape “xxx”	Specifies the conversion for the non-printable characters of the value. The set of values considered “printable” – WordChar - can be adjusted from the default. If not specified the default action is delete. The default set of characters comprises the letters, the numbers, the underscore, the hyphen, the period, the dollar sign.
WordCharsAdd	“xxx”	Adds a set of characters to the list of characters considers a WordChar.
WordCharsDel	“xxx”	Removes a set of characters to the list of characters considers a WordChar.

Automatic SQL Select Generation

Fields may be used in a **`QuerySql=`** statement. These fields offer value interpretation for the columns returned by that SQL query and must be specified in the same order as the fields in the **`QuerySql`** select.

Meta-Update provides a feature to simplify the select statement and to ensure that the order of the fields declared in the field section and columns selected in the SQL query match.

When a field section is used in a **QuerySql** statement, the fields in the field section are concatenated together, separated by commas, so that a select statement can use this symbol for the list of fields following the select.

An additional feature of a field declaration supports the case where a field has an SQL fragment other than the name – for example an inner select.

This may be specified before any field formats and after the field position as **Sql="…"**

An automatic tag and field is assigned with the text of the field names or field **Sql=** text string separated by commas

References within the Sql text is dereferenced when creating the string.

The CTL tag and a field made up of the field section name followed by **-SqlSelect** contains the string.

Here's an example:

```
[QrySql]
QuerySql      =      Q,                                          &
                     SqlFields,                                  &
                     select  $CTL, SqlFields-SqlSelect$          &
                     from    table_x                       A     &
                     where  field1 = '$Arg, some_argument$

[SqlFields]
OBJID         =      $
DATE_CRE      =      $  Sql="(Select date_cre                    &
                            from  table_y                        &
                            where cre2x = A.OBJID) as DATE_CRE"  &
                        Date: yyyy-Mmm-dd
FIELD3        =      $
```

When the **QuerySql** is executed, the text in **$CTL, SqlFields-SqlSelect$** will contain:

```
OBJID , (select date_cre from table_y where cre2x = A.OBJID)
as DATE_CRE , FIELD3
```

Date Fields

ARS supports two different date fields in the database.

- One is called a Date/Time field.
  This is accurate to one second between 1970 and 2034-Mar-23ish.
  For ARS purposes, this is an "epoch" time. It represents a whole number of seconds from January 1, 1970 in Universal or Greenwich Mean Time.

- The other is a Date only field containing no time component.
  For ARS, this is a "Julian" date from an 1, 4713 BC through Jan 1, 9999

When a date is entered into the ARS User Tool, it is considered to be in the local time zone of the machine that is running that User Tool. Meta-Update uses the standard libraries to

convert dates and also presumes that date values are in the locally set time zone of the Meta-Update process.

ASCII file date fields, either from an ARS field, a CSV column, or SQL column, are character strings that Meta-Update must interpret and convert when assigning them to ARS date fields.

Meta-Update can read date data as either one of the above raw “julian” or “epoch” formats, and a normal date specifying the year, month, day, hour, minute, seconds.

By default, a date is specified in any one of the following ways:

yyyy/mm/dd hh:mm:ss
yyyy-mm-dd hh:mm:ss
yyyy.mm.dd hh:mm:ss
yyyymmddhhssmm

One could use this format in making a constant assignment:

```
CreateDate    =      2003/12/31 10:15
```

Any missing components will be treated as if they were zero (one for month and day). These assignments are equivalent:

```
CreateDate    =      2003/01/01 00:00:00
CreateDate    =      2003/01/01
CreateDate    =      2003/
```

These defaults can be overridden on a field-by-field basis when the field is defined in a field section. Simply code the Date format specification when the field is defined. Once a date specification is coded, the file must contain all components specified.

Meta-Update can also accept date data as the raw ARS date/time integer, or raw Date only integer. These are known as an “epoch” and “julian” dates, respectively.

Date formats can be specified in one of three exclusive ways:

Date epoch
Date julian
Date “Date format”;

“Date format” is a string containing the following special symbols and any set of other characters acting as component separators and is terminated by a semi-colon or wrapped in double quotes.

yyyy	a four digit year
yy	a two digit year. Years equal or above 70 are presumed to be offset by 1900, below 70 by 2000.
M	a one or two digit month.
Mm	a two-digit month.
Mmm	a three character month abbreviation (Jan, Feb, etc.)
Mmmm	the full month name (January, February, etc.) Note case difference between minute and month specifications.
d	a one or two-digit day of month.
dd	a two-digit day of month.
ddd	a three-digit day of year (Julian date)

h	a one or two-digit hour. Must be followed by a separator character or must be the end of string.
hh	a two-digit hour. If the hour component is missing, 00:00:00 is assumed.
m	a one or two-digit minute. Must be followed by a separator character or must be the end of string.
mm	a two-digit minute If the minute component is missing, 00:00 is appended to the supplied hour. Note case difference between minute and month specifications.
s	a one or two digit second. Must be followed by a separator character or must be the end of string.
ss	a two digit second If the second component is missing, 00 is appended to the supplied hour.
A or a	the string AM or PM will modify the hour specified. This string is case insensitive.

If a one digit component is specified, it must be followed by a separator or, if not the month, by the end of the value string. The minimum date component must include year month, and day, or year and Julian day. Two digit years are not recommended.

Examples:

Default date format	`Date_Field = $ Date "yyyy/Mm/dd hh:mm:ss"`
OS/390 Julian date	`Date_Field = $ Date "yyddd"`
American date format	`Date_Field = $ Date "Mm/dd/yyyy hh:mm:ss"`
European date format	`Date_Field = $ Date "dd-Mm-yyyy hh:mm:ss"`
ARS Date values	`Date_Field = $ Date epoch`

Numeric Fields

Numeric fields should be specified without thousands separators and with a period as the decimal separator.

If the numeric field in the CSV has a different format, Subst formatting can be used to transform it to the expected format.

For example, say the numeric field in the file is “1.983.217,97” (‘.’ for thousands separators and ‘,’ for the decimal point:

```
Numeric_Field = $ Subst /.// Subst /,/./
```

By applying the above “Substitutes”, in order, the above value is transposed into “1983217.97”.

Similarly, a value of “1,234,567.89” will be transposed into the “1234567.89” by this field format:

```
Numeric_Field = $ Subst /,//
```

Quotes in Field Values

CSV files are generally defined by these rules:

Values are separated by commas.
All lines are terminated by a <lf> or <cr><lf> combination.
Spaces are considered significant.
Values containing a comma must be quoted with the double quote character.
Values containing a double quote character escape that character with another quote.

Meta-Update extends these rules on reading to permit several types of common flaws found in CSV files, no matter how generated. It allows you to specify a different quote and separator character. In addition, with Meta-Update, fields can have embedded new lines. These types of files are difficult to use with the standard tools such as the ARS Import tool and Microsoft Excel.

Finally, Meta-Update can allow some values with embedded un-escaped, or un-doubled, double quotes.

Consider these CSV records:

```
a,b,c,"some text"."some more text
with a new line",,,
```

This is really a single record that is commonly generated from database extracts. Consider this record, similar to the above, but with a field containing a single double-quote character:

```
a,b,c,"some text"."some more text
"with a new line",,,
```

Meta-Update will handle this by establishing the value for the fifth field as:

```
some more text\n"with a new line
```

Consider these CSV records:

```
"val-f1","val-f2 "with embedded and undoubled quotes"","val-f3"
```

In fact the CSV, if it had all embedded quotes doubled, would look like this:

```
"val-f1","val-f2 ""with embedded and undoubled quotes""","val-f3"
```

Meta-Update will assume a doubled quote followed by a separator has a single quote as the last character of the field value. The value for field 2 for either of the above two CSV example would be:

```
val-f2 "with embedded and undoubled quotes"
```

Finally, consider this record where the single quote appears as the character before the end of the line. Meta-Update in this case will assume that that quote terminates the field and the record.

```
a,b,c,"some text"."some more text"
with a new line",,,
```

The value of the field will be:

```
some more text
```

The effect is that this “record” will be truncated and the next record will be read incorrectly.

The command script itself could declare an error in these cases by ensuring a field value that cannot be null is not null in any assignment section. Ensure the field tested follows the field that may contain these values with embedded new-lines and quotes.

```
[Asg]
@Cmd = @if(“$Finp, NON_NULL-FIELD$” == “”)                    &
       Abort, E, Incomplete or badly formed CSV record: $Finp, CSV-
KEY$
. . .
```

Assignment Reference

Assignment Reference

Software Tool House Inc

About Assignment Sections

Assignment sections are where you specify which fields will be updated with what values for any given `Update=` or `Create=` ARS schema and field set.

Assignments specify a target field on the left, an equal sign, and an assignment value on the right. Here as an example used to update a Help Desk ticket:

```
[HpdUpdate-asg-upd]
Status          =       Closed
Work Log        =       "Auto closed in batch process: "
Work Log        =       Arg,  RunName
2400000001     =       "Auto Closed"
```

When Remedy updates or creates a record through the API, it is supplied a list of field – value pairs on the Create, Update, or Merge API call. In the Assignment sections of Meta-Update scripts, you build the lists of field-value pairs that Meta-Update will use on its calls to these API functions.

Assignment sections let you

- Reference sets of pre-loaded records and load other records.
- Create complex fully-nested conditions to control both the values and the fields in the assignments.
- Split data values using pattern matching regular expressions.
- LookUp and translate values using a combination of files, queries, and SQL queries.
- Spawn external processes or ARS Server processes.

Assignments may also be special keywords that allow commands to be specified.

This example will assign all fields in the target schema with the values from the source record (and schema, server, user), with matching field ids. The source record was associated with the Tag, HpdSrc, as a result, of a Query, Load, Update, or Create. These source and target schemas do not have to be the same or even on the same server.

```
[Assignment Section]
@Cmd            =       Copy, HpdSrc, DupIgnore, CoreAssign,            &
                        Skip: field_1, field_31
```

Assignment sections may be broadly classed as

- An update to, or create of an ARS record or an output CSV file record

  These assignment sections are used to specify the field / value pairs that will be used in an update or create to a single ARS record to the form specified in the [Controls] `Create=`, or, `Update=` statements, or a create of additional row of a CSV as declared in the calling section's File= statement.

- Sections with no ARS or CSV targets

  These are specified in the command section to fire at specific times such as Initialization, after the iteration record is loaded, after the output is performed, after launches are performed, and at termination.

  They are used to set script variables and invoke external programs or server processes. This example examines the passed arguments and creates a query that will result in either one record or a range of records.

```
[asg-script-init]
# We should have at least a single id as an argument
```

```
@Cmd     =  @if("$Arg, Id1$ == "")
  @Cmd     =  Abort, E, Usage:  -p id1 [ id2 ] \n                  &
                        where  id1   is the starting id            &
                        and    id2   is the ending id
# Check in our second id was coded
@Cmd     =  @if("$Arg, Id2$ == "")
  @Cmd     =  Ref,  MyVars,  Qry,  "'1' = \"$Arg, id1$\""
@Cmd     =  else
  @Cmd     =  Ref,  MyVars,  Qry,  "'1' >= \"$Arg, id1$\" AND       &
                                    '1' <  "\"$Arg, id2$\""
@Cmd     =  endif
@Cmd     =  Msg, D, Qry: $MyVars, Qry$
```

- String sections

  These are used when a section uses the **`Output=`** to create a pattern file.

```
[asg-pattern]
String   =  "Record Id        $Src, 1$       $Src, 179$"
LoadQ    =  SrcReq, SHR:People, '1' = "$Src, 1$"
String   =  "Requestor Id  .. $SrcReq, 1$      $SrcReq, 179$"
# include pattern file for this language:
File     =  @if("V, Lang$" == "en") asg-pattern-sub-en.ptn
String   =  ""
String   =  "Generated        $time$"
```

Using Assignment Sections

Assignment section names are specified in the command sections with various **Assign=** keywords. They are also specified in **Include** assignment commands.

This example is a command section that specifies three different assignment sections: HpdUpdate-asg-init, HpdUpdate-asg-create, HpdUpdate-asg-upd.

```
[HpdUpdate]
AssignInit    =      HpdUpdate-asg-init
Update        =      HpdTgt,                                  &
                     HPD:Help Desk,                           &
                     '1' = "$Arg, Id$"
AssignNew     =      HpdUpdate-asg-create
AssignNew     =,     HpdUpdate-asg-upd
Assign        =      HpdUpdate-asg-upd
```

The specific **Assign=** keyword or command semantics implies a target.

In the above example, both the **AssignNew=** and **Assign=** sections, **HpdUpdate-asg-create** and **HpdUpdate-asg-upd**, will apply to an ARS record in the HPD:Help Desk form. In these assignment sections, the fields of the HPD:Help Desk form are assigned values and the assignment commands may be used as needed. Similarly, if the section uses an **Output=** to create ASCII files, values will be assigned to the fields.

The **AssignInit=** section, has no target. Only the assignment commands may be used in these assignment sections. There are no fields to be assigned values.

The following keywords may be coded in a control section to specify assignment sections. Any number of sections can be coded with any keyword and the same section may be coded with many keywords.

Each of these keywords specify assignment sections that have no output targets. The different keywords are used to specify the point during the section process that the assignment sections will be used. These assignment sections, because they have no target ARS schema or Output file, only allow the @Cmd and LoadQ "pseudo-fields" as the assigned fields.

AssignInit	Specifies the assignment sections to be applied before processing begins for the section.
AssignTerm	Specifies the assignment sections to be applied after processing ends for the section and the section is about to be closed.
AssignPre	Specifies the assignment sections to be applied directly after loading the section's record. This is applied before any subsequent Loads, Updates, or Assignments.
AssignPost	Specifies the assignment sections to be applied directly after completing any updates but before any launches and the next iteration of the section. This is applied whether an error occurred or not.
AssignPostOk	Specifies the assignment sections to be applied directly after completing any updates but before any launches and the next iteration of the section. This is applied whether only when an error did not occur in the update or any launches. .This is applied before the AssignInitPost sections.

AssignPostErr	Specifies the assignment sections to be applied directly after completing any updates but before any launches and before the next iteration of the section. This is applied whether only when an error did occur in either the update or any launches. .This is applied before the AssignInitPost sections.
AssignPostLaunch	Specifies the assignment sections to be applied directly after all launches are invoked, if any, but before the next iteration of the section. This is applied whether only when an error did occur in either the update or any launches. .T

Each of these keywords specifies assignment sections that have either and ARS record or an output file, record or pattern output targets.

Assign	Required for all sections that have output - **Update=, Create=, Output=** keywords. Specifies the assignment sections to be applied when an update record is found or a new ARS or file record is to be created.
AssignNew	Only used with an **Update=**. Optional. Specifies the assignment sections to be applied when no update record is found. This will cause the **Update=** to create a new record if the **Update=** query returns no records.
AssignOpen	Only used with an **Output=**. Optional. Specifies the assignment sections to be applied when the file is first opened, or for pattern files, when the next file is being opened.
AssignClose	Only used with an **Output=**. Optional. Specifies the assignment sections to be applied when the file is closed (at the job end), or for pattern files, when the current file is being closed just before the next file is opened.

Assignment Targets

Assignment sections are applied to target ARS records, target columnar files, target pattern files, or with no target at all.

The general format of an assignment is:

```
[Assignment Section]
Target_Field =      "some value"
```

For ARS records, the "Target Field" is an ARS Database Field Name, or Field ID belonging to the target schema.

```
[HpdUpdate-asg-upd]
Status         =      Closed
Work Log       =      "Auto closed in batch process: "
Work Log       =      Arg,  RunName
2400000001    =      "Auto Closed"
```

For columnar Output Files, the "Target Field" is a field as defined by the file definition section in this script.

```
[FleDef-out-csv]
Type           =      Delimited, ",", FldHdr
Fields         =      FleDef-out-csv-flds
[FleDef-out-csv-flds]
Case ID        =      $
Diary_User     =      $
Diary_Date     =      $      Date: yyyy-MM-dd
Diary_DateEpoch  =    $      Date: epoch
Diary_Text     =      $

[HpdWorkLogReport-asg-upd]
Case ID        =      HpdSrc,  1
Diary_User     =      HpdSrcDiary,  User
Diary_Date     =      HpdSrcDiary,  Date
Diary_DateEpoch  =    HpdSrcDiary,  Date
Diary_Text     =      HpdSrcDiary,  Text
```

For Output Pattern Files, the "Target Field" are the reserved words **`String=`** and **`File=`**. This is also called a String target and can be used in the Reference assignment command to build complex strings.

```
[HpdWorkLogReport-asg]
String =      Case ID:             $HpdSrc,  1$
String =      Diary_Entry by:      $HpdSrcDiary,  User$
String =                   on:     $HpdSrcDiary,  Date$
String =      $HpdSrcDiary,  Text$
String =      ""
```

For any of the above target and for assignment sections having no output targets, a "Target Field" may be "**`@Cmd`**". This is aspecial keywords that allow commands to be specified.

```
[Assignment Section]
@Cmd           =      Copy, HpdSrc, DupIgnore, CoreAssign,          &
                      Skip: field_1, field_31
```

The above example is only allowed when assignment section has a target and that target is one of: an ARS schema, a columnar file, or a string. It will copy all matching fields except fields field_1 and field_31 and any fields not defined in the target.

Section Target Types

There are three different types of assignment sections which differ in their targets:

ARS or CSV	Target is an ARS Record in the schema declared in the **`Create=`** or **`Update=`** keywords, or the @ars Reference assignment command, or a columnar File record whose fields are declared in the **`Output=`** keyword. These sections can only be specified by `Assign=, AssignNew=, AssignOpen=, and AssignClose=` command section keywords.
No Target	There is no output target for these assignments. There is no ARS or record to create or update or output file record to create. These sections are used to assign script variables, to invoke external processes, issue messages, abort an operation, and so on. These sections can only be specified by `AssignInit=, AssignTerm=, AssignPre=, and AssignPost= AssignPostOk= AssignPostErr=,` and `AssignPostLaunch=.`
Tag Target	Called by a Reference command, the target is a single string reference tag. Any fields assigned in these sections become a field of the Tag that this section was called for. See “Assigning a set of fields and values to a single Tag” in the Assignment Command, Reference command.
Pattern Target	The output target for these assignments is either: ➢ a named string specified with the Reference assignment command, ➢ or a pattern file “record”, specified with the **`Output=`** keyword. There are only two “fields” available as an assignment’s target field. These are **`String=`** and **`File=`**. Each **`String=`** assignment specifies a single line of text and is automatically terminated with a new line. If only a single **`String=`** is encountered in an assignment process, then the new line is not implied. A **`File=`** specifies an ASCII text file to be read and copied into the target named string or pattern file “record”. Any Meta-Update references in the body of the file are resolved.

Assignments

The assignment section specifies the Remedy fields to be updated and the values to update those fields with.

The assignment section can also `Load=` and `LoadQ=` statements. These are processed after the main section's entire `Load=` statements are processed and can use the record loaded by the `Query=` or `File=` or `Update=` statement. In the assignment section, these are processed in the order encountered.

An assignment section consists of keywords which are Remedy Field Ids or "Database Field Names" in the target schema - the one being updated, or file fields in an output columnar file, or the special target field names: String and File for output pattern files.

```
Field          =      Assignment Value
```

Processing an assignment for the same field twice causes concatenation as in this example. It only makes sense for text and diary fields.

```
Status         =      Closed
Work Log       =      "Auto closed in batch process: "
Work Log       =      $Arg, WkTxt
2400000001     =      "Auto Closed"
Close Date     =      $DATE$
```

Note that values must be compatible with the Remedy fields if the target is a Remedy record.. Enum values can be specified as a numeric value or the Remedy enum label.

The assignment value can be a constant, or a simple reference.

Assignment Value := Constant | Keyword | Parameter | Environment | Reference

Constant := number | string | quoted string

Keyword := Keyword values supported:

$NULL$	applies to all Remedy data types and is an assignment to NULL. Causes deletion of attachment fields' attachments.
$TIMESTAMP$	gives the current date and time and is only available for use with a Remedy time field.
$DATE$	is equivalent to $TIMESTAMP$.
$DAYEND$	current date at 23:59:59
$DAYSTART$	current date at 00:00:00

If the value cannot be interpreted, a warning is issued and the value is interpreted as a string constant.

Parameter := Deprecated.
The n'th parameter passed on the command line
$nnn $nnn If nnn exceeds the number of parameters available, an error is thrown and the update does not proceed.

Note that there must be exactly three digits following the $.

Named parameters are self-documenting and easier to use. These are simply a string Reference with the Tag being Arg by default and the variable name being the parameter defined by the Arg= value of the Main section.

Environment := Deprecated.
An environment variable set before Meta-Update is executed.
$aaaaa If the named variable (aaaaa) is defined in the environment, its text value is used. If it is not found, a warning is issued and the $aaaaa is used as Keyword missing the terminating $.

The Environment is available as a reference in the pre-defined tag: **ENV**. To assign an environment variable's value, simply use the reference form, as in this example:

```
Path        =     ENV,  Path
```

Reference := Tag , Field
Tag is a previously loaded ARS or file record, or a named collection of strings. Field is the name or ID of one of the Tag's form's fields.

The field in references does not have to be of the same type as the field being assigned. It must be of a compatible type. If a conversion is not possible, the operation fails.

Field := Field Id | Field Name

Field ID := a long numeric Remedy Field Id

Field Name := The Remedy ARS Field database name

Using $NULL$ is not the same as not assigning a field. Not assigning a field allows Remedy to choose that field's default value on a create record. On an update, the value is not changed.

Diary entries are always appended to on an update.

Conditional Assignments

Any individual assignment or can be made conditional by preceding the assignment value by an `@if()` construct. This includes the special assignment commands.

There are two styles of @ifs:

```
Field1  =  @if (exp) assignment
Field2  =  @if (exp, true-value, false-value)
```

In addition, a full structured if facility is provided.

Assignments can use either format. If the first format is used, and the expression evaluates to false, the assignment is not made at all. If the expression evaluates to true, the assignment to the field is made.

In the second format, an assignment to field2 is always made. If the expression is true, the true-value is assigned. If false is false, the false-value is assigned.

Assignment commands can only use the first format. If the expression evaluates to false, the command is not processed.

```
@Cmd      =  @if (exp) assignment command
```

In the second format, the values can also be @if constructs.

The expression syntax is as follows:

op-not:	!	Boolean not
op-bool:	&&	Boolean and
	||	Boolean or
op-rel:	== != > >= < <= ~=	case sensitive
	==~ !=~ >~ >=~ <~ <=~ ~=~	case insensitive
val:	"string"	
	digits	
	(val)	
rel-exp:	val [op-rel val]	
	(rel-exp)	
exp:	[op-not] rel-exp [bool-op rel-exp]	
	(exp)	

Values compared are string references and are case sensitive. The NULL value compares equal no matter if it is specified as the $NULL$ keyword or an empty string.

The operators ~= and ~=~ are leading string compares. If the left string begins with the right string, the expression yields true. For example:

```
"abcdef" ~=  "abc"          yields true
"abcf    ~=  "abcdef"       yields false
"abcdef  ~=  "A"            yields false
"abcdef  ~=~ "A"            yields true
```

A string containing all digits is converted into a number when being compared against a number or when being used as a Boolean value by itself. A zero-length string by itself is considered false.

Examples:

```
Status        =      @if("$Xn, XnCode$" == "Delete") Inactive
Status        =      @if("$Xn, XnCode$" != "Delete") Active

Status        =      @if("$Xn, XnCode$" == "Delete", "Active",
"Inactive")
```

The above statements set the status to Inactive if the incoming transaction code is equal to "Delete". Otherwise, the Status is set to Active.

The incoming transaction code is read from a File statement with a tag of "Xn". The field within that file is "XnCode".

Example:

```
Notes         =      "Notes\n"
Notes         =      Xn, Notes
Notes         =      @if("$Xn, Notes2$" !== "") "\n===========Notes
2\n"
Notes                Xn, Notes2
```

Or, with a little "improvement":

```
Notes         =      "Notes\n"
Notes         =      Xn, Notes
Notes         =      @if("$Xn, Notes2$" !== "",                    &
                         "\n===========Notes 2\n$Xn, Notes2$",      &
                         "No notes2 data\n"                    )
```

If the Notes2 field of the Xn record is empty, the Notes field will contain the text Notes and the value of the Notes field in the transaction followed by the text, "No notes data".

```
Notes
Value from Xn, Notes
No notes2 data
```

If it is not empty, the Notes field will look like:

```
Notes
Value from Xn, Notes
===========
Notes 2
Value from Xn, Notes2
```

IF Statement

The structured, nested, if facility is like the if of many programming languages such as c, Java, and so on. Basically,

```
if ( condition )
    condition true assignments ...
else
    condition false assignments ...
    if ( condition2 )
        Condition2 true assignments ...
    else
         condition2 false assignments ...
    endif
```

```
endif
```

Meta-Update uses the first format of the assignment conditional “if“, followed by no value to specify an IF statement. See Conditional Assignments above for more information on specifying the conditional expression.

To distinguish this if from an assignment, the special target field @Cmd is used. So,

```
@Cmd          =      @if ( condition )
   real ARS target field assignments
   …...
@Cmd          =      else
   real ARS target field assignments
   …...
@Cmd          =      endif
```

An “else” can follow and an “endif” terminates the “if”. If the expression evaluates to true, all assignments up to the “else” are processed. If false, all assignments are skipped until the “else” and all subsequent assignments are processed.

Ifs can be nested as needed up to a maximum nesting level of 100.

This following assignment sections are entirely equivalent:

```
Notes         =      “Notes\n”
Notes         =      Xn, Notes
@Cmd          =      @if(“$Xn, Notes2$” !== “”)
    Notes     =          “\n===========Notes 2\n”
@Cmd          =      else
    Notes     =        “  No notes2 data”
@Cmd          =      endif
```

and

```
Notes         =      “Notes\n”
Notes         =      Xn, Notes
Notes         =      @if(“$Xn, Notes2$” !== “”,                    &
                         “\n===========Notes 2\n$Xn, Notes2$”,      &
                         “No notes2 data\n”                    )
```

LookUp Assignments

LookUp assignments allow a data value to be translated without having any extra ARS tables.

```
@LookUp [,]    Section,    Src Value
```

`Section` Specifies the LookUp section that gives the source and target values for the look up. It also specifies any default value and match failure options. See LookUp Sections below for more information.

`Src value` This is the value that will be looked up. It is a string reference. If this value is found in the LookUp section, the value associated will be assigned.

If this value is not matched, a default value can be assigned, the assignment to the field can be skipped, or the processing for this update

can be aborted. These actions are specified in the LookUp section's NoMatch statement.

Like all assignments, a LookUp can be made conditional.

Example:

```
Status        =      @LookUp, AsgLookUp, $Xn, Status$

[AsgLookUp]
Active        =      Current
Deleted       =      Inactive
Cancelled     =      Inactive
```

Assignments Commands

Special commands can be included in assignment sections. These are identified with the field name "`@Cmd`"

Like all assignments, these special commands can be conditional.

```
@Cmd     =   Copy, ...
@Cmd     =   @if(...)                          &
             Copy, ...
```

Assignment commands allow

- assignments to script variables of strings, Remedy records, the current environment
- if conditionals to be coded
- external processes to be spawned on the client or the server
- messaging, aborting
- copy like fields into the target
- conditional breakpoints when debugging is enabled

The following commands can be used:

Copy	Copy all like fields from one record into the target record.
Include	Process another section of assignments
Abort	Abort the update or output.
Msg	Issue a message.
Spawn	Spawn an external process
Reference	Assign a new string reference
Break	Execute a debugging Breakpoint

@if	allows nested ifs
else	
endif	

Assignment Commands

Assignment commands allow

- assignments to script variables of strings, Remedy records, the current environment
- if conditionals to be coded
- external processes to be spawned on the client or the server
- messaging, aborting
- copy like fields into the target

Copy Command

The Copy command is used to copy all fields that have matching field Names or Ids from one loaded record into the target update record. These do not have to be from the same schema or the same server, or even ARS records.

```
@Cmd   = Copy, SrcTag,
           [, {  DupAppend      | DupOverwrite | DupIgnore     } ]
           [, { MismatchIgnore | MismatchWarn | MismatchError } ]
           [, { CoreAssign      | NoCoreAssign                } ]
           [, @if(exp) ]
           [, SkipDisplayOnly ]
           [, SkipNull ]
           [, Skip:   fld [ , .. ] ]
           [, Fields: fld [ , .. ] ]
```

Targets of Copy commands

The Copy command can be used in assignment sections for the following targets:

ARS records — in assignment section called from **`Update=`** or **`Create=`** command sections, or from **`Reference @ars`** commands called from other assignment sections.

File records — in assignment sections called from command sections containing an **`Output=`** for a columnar file (delimited or fixed).

String patterns — in assignment sections for an output pattern file, or the assignment reference command:

```
@Cmd    =   Reference,     Tag,   Var,  @Sec
```

A copy command in **`[Sec]`** will iterate though all fields from the source tag building a single string based on the supplied **`Ptn`** value (required).

String Tags — through the use of the assignment command:

```
@Cmd    =    Reference,   Tag,   @,    Sec
```

A copy command in **`[Sec]`** will copy fields from the source tag creating like named fields in the specified **`Tag`**.

Defaults: **`DupIgnore, MismatchWarn, NoCoreAssign`**

In the case where the target is a String Tag, the duplicate assignment option default is **DupOverwrite**.

To use **DupIgnore** on String Tag targets so that prior assignments may be made, the following command should be run at an opportune point (the **AssignInit** or **AssignTerm** sections for example) to remove the last iteration's values:

```
@Cmd      =   Reference,   Tag,     /del
```

Only one of **`Skip:`** or **`Fields:`** can be used and it must be the last keyword on the command.

Keywords for Copy commands

`SrcTag`	Specifies the source reference tag. This can be a string reference which must evaluate to a defined tag. This Tag can be a loaded Remedy record, a loaded File record, or a String Tag. Only fields with the same Ids (if both the source and target are Remedy records), names and types can be copied. Options indicate what to do on mismatches. You can also override the copy of some fields by making explicit assignments to those fields ***ahead*** of the copy command and selecting the default option: `DupIgnore`.
`DupAppend`	If a field already has been assigned a value, this will cause the copied value to be appended to the current value if possible. This is only possible for character and diary fields.
`DupOverwrite`	If a field already has been assigned, this will take the value in the copy source record.
`DupIgnore`	If a field already has been assigned, this will ignore the value in the copy source record. If not specified, `DupIgnore` is taken as a default.
`MismatchIgnore`	When comparing the source schema to the target schema, this will ignore any mismatched fields.
`MismatchWarn`	When comparing the source schema to the target schema, this will ignore any mismatched fields and issue a warning message.
`MismatchError`	When comparing the source schema to the target schema, any mismatched fields will cause an error and the assignment will not be processed. The record will not be updated.
`CoreAssign`	Indicates that the "core" fields are also to be copied. This is only useful when doing a Merge operation. Note that specifying CoreAssign will also cause the Request Id field to be assigned. If you do not want this, specify Skip: 1 in addition to the CoreAssign.
`NoCoreAssign`	Indicates that the "core" fields are not to be copied. This is the default. **`CoreAssign`**, **`NoCoreAssign`**, and the **`MisMatch`** options apply only if the target is a Remedy record.

`Ptn "string"` Specifies an pattern text string that will be used for each field and value being copied. Required and used only when the target is a single pattern.

In the pattern string, the following extra substitutions can be made:

`@FldSrc`	***$TagSrc, Field Name$***	(the value)
`@FldNme`	`Field Name`	(the field's name)

`@if(exp)` Specifies an expression to be applied to each field being copied. If the expression evaluates to true, the field is copied and an assignment is made to that field. If the expression evaluates to false, an assignment to that field is not made.

In the expression, the following extra substitutions can be made:

`@FldSrc`	`$`***TagSrc***`, Field Name$`	(value)
`@FldTgt`	`$`***TagTgt***`, Field Name$`	(value)
`@FldNme`	`Field Name`	(field's name)

TagSrc	is the SrcTag on the copy command.
TagTgt	is the Tag on the Update statement.

A field is the **TagSrc**'s field name being worked on. This iterates through all data fields of **TagSrc**.

`SkipDisplayOnly:` Specifies that ARS Display Only fields are ***not*** to be copied. Ignored when the **SrcTag** (the source tag) is not a Remedy record. May be abbreviated to **SkipDO**.

`SkipNulls:` Specifies that source fields which have a **$NULL$** value are ***not*** to be copied

`Skip:` If used, must be the last keyword on the command. **Skip:** is followed by a comma separated list of fields to ignore from **SrcTag** (the source tag).

Specifies a list of fields that are ***not*** to be copied. The field names are fields of the source tag.

Field Ids may be used only if the source Tag refers to a Remedy record.

If you are doing a Merge, and especially if you are copying data from another server, you may not want the **Request Id** field (field **1**) assigned. Specify **Skip: 1** to avoid this.

`Fields:` If used, must be the last keyword on the command. **Fields:** is followed by a comma separated list of fields to copy from **SrcTag** (the source tag).

Specifies a list of fields that ***are*** to be copied. The field names are fields of the source tag.

Either **Skip:** or **Fields:** can be used but not both.

CoreAssign and Core Fields

The meaning of "Core fields" as applied to the copy command, does not include core fields that can be assigned without the use of Merge. That is, a Copy command with the default **`NoCoreAssign`** option, ***will*** copy these fields:

Field Id	Common Field Name
4	Assigned To
7	Status
8	Short Description

To not assign these fields with the copy command, add them to the Skip list.

This option is ignored when the target is not an ARS record, and all fields that can be copied – including core fields in the source if that source is an ARS record – are copied.

Examples of Copy Commands

When merging two different records, it is often desired to not overwrite the contents of a field with $NULL$. We also do not want to replace the Diary log with that of the source. We will instead add a new Diary entry indicating the records were merged. Note that we are not using a Merge operation but a normal Modify. These assignments, including the copy command will do that.

```
@Cmd     =   Copy, Src, @if("@FldSrc" == "" && "@FldSrc" != "")
```

In this example, we are copying from another server but we do not want the request id copied.

```
@Cmd     =   Copy, Src, CoreAssign, Skip: 1
```

This example could be an assignment section for a pattern file to build an HTML table from all the fields.

```
String   =   <table><tr><th>Field</th><th>Value</th></tr>
@Cmd     =   Copy, Src,                                          &
             Ptn  "<tr><td>@FldNme</td><td>@FldSrc</td></tr>"
String   =   </table>
```

Include Command

The Include command is used to include assignments from another section. This is in addition to the set of assignment sections listed in the control section's `Assign=` or `Update0=` keywords.

```
@Cmd     =   Include,    Section
```

`Section` This is the section name to be included.

It can be a constant

```
asg-BaseElement
```

If the section name does not exist, an error will be thrown.

It can also be a string reference expression:

```
"Sec-Upd$Src, Typ$"
```

When the included section name is derived from a string reference expression, the resulting section name does not need to exist. If it doesn't, no section will be included.

If the whole command is prefixed with an **`@if(exp)`**, that expression must evaluate to true or no new sections will be included.

Abort Command

The Abort command is used in an assignment section to discontinue an update or to simply issue a message. It is generally made conditional. Options include the severity of the error to generate and a message to be written to the trace file. The IdLog, if being created, reports the operation status as "Aborted".

```
@Cmd    =  Abort   [ , Severity  [, { Launch | Msg } ]  [ ,  Message
] ]
```

`Severity`

D	Debug	No error is reported. A Debug message is produced in the trace files. These are normally inhibited.
I	Informational	No error is reported. An Information message is produced in the trace files.
W	Warning	No error is reported. A Warning message is produced in the trace files.
E	Error	An error is reported. An Error message is produced in the trace files. Processing of the control section stops for this record and no other sections are launched for this record.

Note that if the Severity is Error the message appears as an error but no actual error condition is raised. Also note that if the message severity is Debug, debugging logging must be turned on for the message to appear.

`Launch` This is a keyword that must be coded as is. If coded, any Launches coded will be executed. This is not the default behaviour. Note that the record being created or updated cannot be reread (as the update was aborted) and so there should be no references to the Update tag in any Launched sections.

`Message` Any string including any string references.

Defaults **`I, User Abort taken for $CTL, Operation$ on $CTL, Schema$ ID: $CTL, ID$`**

AttachSave Command

The AttachSavecommand allows you to save an attachment reference on the machine that Meta-Update is running on.

```
@Cmd    =  AttachSave,    Tag,    Fld,    FileName
```

`Tag` This is the Tag containing the ARS record that will have an attachment field that needs to be saved.

`Fld`	This is the field name or id of the attachment field.
`FileName`	This is a reference string that will be the name of the file.

Examples:

```
@Cmd    =  AttachSave,    Src,    Attach1,    $Src, Attach1$
```

In the above example, the attachment field itself is used as an output file name. Note that this will fail if the path information is incorrect or the paths do not exist.

You can use regular expressions to remove all path information as needed.

Msg Command

The Msg command is used in an assignment section to produce a message. It has no other effects. It can be made conditional. Options include the severity of the message to generate and text of the message to be written to the trace file.

```
@Cmd    =  Msg   [ , Severity  [ ,  Message ] ]
```

`Severity`

D	Debug	A Debug message is produced in the trace files. These are normally inhibited.
I	Informational	An Information message is produced in the trace files and may be echoed to the console.
`W`	Warning	A Warning message is produced in the trace files.
`E`	Error	An Error message is produced in the trace files but no real error condition is raised.

`Message` Any string including any string references.

Spawn Command

The Spawn command allows you to launch a separate process. Any valid executable can be coded. The process must complete for Meta-Update processing to continue.

```
@Cmd    =  Spawn,   command
```

`command` This is any string that can be de-referenced and makes sense for the operating system that Meta-Update is being run on.

The process should return 0 to indicate success and any non-zero value to indicate failure.

If the spawn itself fails, that is, the operating system will not or can not launch the specified process, the Meta-Update will throw an error, and the assignments will fail.

If the spawn succeeds, but the spawned process returns a non-zero return value, Meta-Update will issue a Warning but continue the assignments.

When a Spawn command is used in an assignment, and that Spawn succeeds, these **CTL** references are automatically set:

CTL	**Spawn_Cmd**	The executed command. This reference is set when any Spawn is encountered.
CTL	**Spawn_rc**	The executed process return code. This reference is set to "-1" when a Spawn is encountered and set to the spawned processes return code when the spawned process completes.

These variables need to be assigned to be saved.

Reference Command

The Reference command allows you to assign a value into a named string reference. That named string reference can be used anywhere any reference can be used after it has been assigned.

This is a very powerful facility allowing you to implement more complex batch functions with Meta-Update.

A string reference is similar to a Remedy record. The Tag identifies a set of named values akin to "fields".

So, if you assigned the value **"xyz"** to a field called **Text** in a Tag called **MyVars** you would get **"xyz"** from this reference: **"$MyVars, Text$"**.

Each Reference command assigns one or more such named values to tags, or can refer to an assignment section where the left hand targets of assignments become named values.

The general format of the Reference command is this:

```
@Cmd    =  Reference,  Tag,   Name,  Value
```

The word "**Reference**" may be abbreviated to "**Ref**". The parts of a reference command are:

Tag	This is the name of the string reference to be assigned. Any name can be used. This is the "**Tag**" that this collection of named values will be referenced by. Use different Tags to group different information in more complex scripts. The Tag itself may be a reference which allows more complex scripts such as configuration driven scripts using arrays of loaded data.
Name	This is the name of the individual "field" within this "Tag". References to this Tag and Field Name will return the value assigned, for example, **$Tag, Name$**. The field may also be a reference.

Some forms of reference commands assign multiple field names to a tag. These are identified with a special reserved keyword for the name.

Name Keyword	Usage
`@`	When the name is specified as a single at sign "`@`": the value following the reference is treated as an assignment section. Any field assigned a value in that section, is set as a reference under the Tag specified.
`@info`	Specific named fields and values are assigned to the tag that give information about another Tag and Field reference. This can be used for example to determine if a field exists in a form.
`@date`	Specific named fields and values are assigned to the tag that gives information about a date value. This can be used for example to determine the name of the day for a given date.

`Value` This is a single value to be assigned. Any outer quotes are removed and references are resolved.

This value may be interpreted differently when special @keywords are used for the Name being assigned, as described above.

When assigning a value to a single named field of a tag, additional functions are available to derive the value. These are identified by a reserved keyword in the value field.

Value Keyword	Usage
`@section`	When the value is specified as a single at sign "`@`": the value following the reference is a "String pattern" assignment section. See Assignment Targets above. The specified "String target" assignment section has only two fields available: `String=` and `File=` and is used to build a single string value that will be assigned to the specified tag and name.
`@LookUp`	The `@LookUp` keyword is followed by the LookUp section and the value look up. See LookUp Assignment above.

@if(c, t, f)	The condition will be evaluated and the true or false value will be assigned to the tag and name.
@eval	This is used to evaluate an arithmetic expression. It can be followed by the keyword "real" to override integer arithmetic and specify that floating-point arithmetic be used. See Using Arithmetic Expressions below.
@fmt **@fmtout**	This is used to transpose a value according to a field formatting string. You can use this to change case and perform substitutions for example. See Field Formats above for more information.
@ars	This is used to create an in-memory ARS record from the specified schema. The name field is interpreted as an assignment section. When needed for an update, this in-memory ARS record can be assigned to an update record.
@val	This is used to do a "double dereference" so that the tag and name are themselves references. This is similar to using **@info** for the assigned name as described above but only extracts the value of the references.
@regex	This is used to apply a regular expression to a value and perform substring extraction from that expression. The Name field is interpreted as a field section to name the extracts substrings. If matched the Tag will contain all fields of this section. The special name **@rc** is assigned 1 indicating the regex was matched or 0 indicating it was not. The Name may be specified as @na indicating that fields are not defined and numerical references will be assigned to the tag for extracted strings. Regular expressions may themselves have references. Meta-Update uses PCRE for evaluating regular expressions.

Examples of Tags and Names:

```
MyVars,                     DoExtraWorkLogRecord
MyVars,                     SkipAuditLog
MyVars                      i
rt-$FleCfg, Schema$,        RequestIdField
rt-$MyVars, i$ ,            TotalRecs
```

Note that in the last two lines the Tag being assigned is dereferenced. Examples could be "**rt-HPD:Help Desk**" and "**rt-9**". The name is fixed making an array or hash of such fields. These can then be looped through as needed. .

Types of Reference commands

A single value is assigned to a Tag and field name that can be referenced as **$Tag, Name$**.

Assigning a single value to a named string reference

```
@Cmd    =  Reference,  Tag,   Name,  Value
@Cmd    =  Reference,  Tag,   Name,  @LookUp, ..
@Cmd    =  Reference,  Tag,   Name,  @if( ..)
@Cmd    =  Reference,  Tag,   Name,  @ Section
@Cmd    =  Reference,  Tag,   Name,  @eval,  [real,]  Value
@Cmd    =  Reference,  Tag,   Name,  @fmt[out],  Value,  Format
@Cmd    =  Reference,  Tag,   Name,  @fmtqry,    Value,  SrcTag
@Cmd    =  Reference,  Tag,   Name,  @val,   SrcTag,  SrcFld
```

Assigning many named values to a single Tag

"Value-Sec" is an assignment section where "fields" are assigned to this Tag.

```
@Cmd    =  Reference,  Tag,   @,  Value-Sec
```

@info assigns a set of fields to describe **$SrcTag, SrcFld$**.

```
@Cmd    =  Reference,  Tag,   @info,  SrcTag,
SrcFld
```

@date assigns a set of fields to describe **a single date value.**

```
@Cmd    =  Reference,  Tag,   @date,  SrcRef
```

ARS Record in-memory

```
@Cmd    =  Reference,  Tag,   Name,  @ars,    schema
```

Assigning a new Tag as equivalent to an existing Tag:

```
@Cmd    =  Reference,  Tag,   Name,  @equ
```

Assigning the result of a special server $PROCESS$ call:

```
@Cmd    =  Reference,  Tag,   Name,  @exec   process  [ arguments ]
@Cmd    =  Reference,  Tag,   Name,  @guid   [ prefix ]
```

Assigning the results of a regular expression applied against a target string:

```
@Cmd    =  Reference,  Tag,   Name,  @regex   regex,  Value
```

Assigning the result of a client spawned process to fields: rc, stdout, stderr:

```
@Cmd    =  Reference,  Tag,   @spawn,   process  [ arguments ]
```

Removing previously assigned references:

```
@Cmd    =  Reference,  Tag,   Name,  /delete
@Cmd    =  Reference,  Tag,   /delete
```

Single value to a Tag string reference:

```
@Cmd     =  Reference,  Tag,   Name,  Value
@Cmd     =  Reference,  Tag,   Name,  @val,   SrcTag,  SrcFld
@Cmd     =  Reference,  Tag,   Name,  @LookUp, ..
@Cmd     =  Reference,  Tag,   Name,  @if( ..)
@Cmd     =  Reference,  Tag,   Name,  @ Section
@Cmd     =  Reference,  Tag,   Name,  @eval,  [real,]  Value
@Cmd     =  Reference,  Tag,   Name,  @fmt[out],  Value,  Format
@Cmd     =  Reference,  Tag,   Name,  @fmtqry,    Value,  SrcTag
```

Each of the above forms can be used to yield a single string variable being set with a value.

```
@Cmd     =  Reference,  Tag,   Name,  Value
```

`Value`

In this simplest form, the Tag, Name reference is assigned the dereferenced **`Value`** text as specified in the reference command.

For example

```
@Cmd = Ref, K, False, 0
```

This will cause references like `$K, False$` to return “0"

Or

```
@Cmd = Ref, K, Ftmp-Nme,                                    &
       "$ENV, tmp$\\$CTL, ScriptFx$-$CTL, Pid$-inp.txt
```

This create a temporary file name in the form

```
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\MyScript-11582-inp.txt
```

when running a script called MyScript under the Process Id 11582.

```
@Cmd     =  Reference,  Tag,   Name,  @val,   SrcTag,  SrcFld
```

Use this to extract a value from a dynamic **`SrcTag`** and **`SrcFld`**. Both of these and be string expressions. This is an alternative to using the `@info` reference command, when the **`SrcTag`** and **`SrcFld`** are known to exist and you are only interested in the value.

```
@Cmd     =  Reference,  Tag,   Name,  @LookUp, ..
```

Use this to perform a LookUp (possibly loading a record) and load the returned value into the Tag and Name field. See LookUp Sections below.

```
@Cmd     =  Reference,  Tag,   Name,  @if( ..)
```

Use this to assign a value conditionally.

The if can be of these two different formats:

```
@Cmd    =  Reference,  Tag,   Name,  @if(exp)  TrueValue
@Cmd    =  Reference,  Tag,   Name,  @if(exp,  TrueValue, FalseValue)
```

In the first case, if the expression is false, no assignment is made to **`Tag`** and **`Name`**. If **`Tag`** and **`Name`** were not defined, they would still be undefined.

In the second case, an assignment is always made to **`Tag`** and **`Name`**.

```
@Cmd    =  Reference,  Tag,   Name,  @ Section
```

The section is a string pattern assignment section.

A string assignment section comprises the normal **`@Cmd`** keywords as well as the special keywords **`String=`** and **`File=`**. These special assignment sections can be used to build long strings. Each separate `String=` is implicitly terminated with a new line character with the exception of a single **`String=`** assignment. New lines from pattern files are copied as is.

The assignment value for the **`String=`** keyword taken as a string reference. Example:

```
String="The ID of the record is      \t$MT, ID$"
String="The submitter is             \t$MT, Submitter$"
```

The assignment value for the `File=` keyword is a file specification valid for the OS. This file contains simply the text of the string with substitutions as above. Example:

```
File      =  ./pattern.txt
```

The file `./pattern.txt` contains:

```
The ID of the record is       \t$MT, ID$
The submitter is              \t$MT, Submitter$
```

```
@Cmd    =  Reference,  Tag,   Name,  @eval,  [real,]  Value
```

This assigns the result of an arithmetic expression to **`Tag`** and **`Name`**. See Arithmetic Expressions below.

```
@Cmd    =  Reference,  Tag,   Name,  @fmt[out],  Value,  Format
```

This assigns the result of a field format applied to a value to **`Tag`** and **`Name`**. See Formatting Values below.

```
@Cmd    =  Reference,  Tag,   Name,  @fmtqry,    Value,  SrcTag
```

This takes the string "**`Value`**" and replaces fields between dollar signs with those found in the **`SrcTag`**, and assigns the new string to **`Tag`** and **`Name`**. Field IDs can be used if the **`SrcTag`** is a Remedy record.

If a field is not found, it is not replaced.

The following example:

```
@Cmd    =  Reference,  Tag,   Name,  @fmtqry,          &
           "'Incident Number' = \"$Incident Number$\"", &
           Src
```

assigns a string to **`Tag`** and **`Name`**. like this:

```
'Incident Number' = "INC-CAL00010021"
```

Incidentally, the following command assigns the same string:

```
@Cmd    =  Reference,  Tag,   Name,  "'Incident Number' &
            = \"$Src, Incident Number$\""
```

However, when Value is a not a constant but a reference such as from a configuration, the **`@fmtqry`** is needed.

Many values to a Tag:

```
@Cmd    =  Reference,  Tag,   @,  Sec
```

`@` When the Name is a single "@", it tells Meta-Update to treat the value as an assignment section where any field can be assigned a value. All fields become a field of the named Tag.

This is a good way to assign many fields to a single Tag in one section.

`Sec` This is the single section name that will be executed. All new fields are added to the Tag. This can be a reference.

Assigning a field twice causes concatenation. If you need to initialise specific elements of the Tag or delete the Tag before this reference command.

After either of these two assignment sections is executed, the following references will be defined:

```
$V, v1$     v1-val
$V, v2$     v2-val
$V, v3$     v3-val
```

```
[Asg]
@Cmd  = Ref, V, v1, "v1-val"
@Cmd  = Ref, V, v2, "v2-val"
@Cmd  = Ref, V, v3, "v3-val"
```

```
[Asg]
@Cmd  = Ref, V, @, Asgv1

[Asgv1]
v1    = v1-val
v2    = v2-val
v3    = v3-val
```

Reference Information — assigning a set of values using references for a Tag and Field

The **@info** command assigns a specific set of fields describing the single reference Tag and Field that is passed to it.

```
@Cmd    =  Reference,  Tag,   @info,  SrcTag,  [ SrcFld ]
```

`@info` — @info requests a specific function. The **SrcTag** and **SrcFld** can themselves be references. @info causes the **Tag** to be assigned a specific set of fields depending of reference passed.

`SrcTag` — this can be a Tag loaded in your script, or a reference that will evaluate to a Tag that is loaded in your script. If only the `SrcTag` is supplied, only those names appropriate to the `SrcTag` "record" that is loaded.

`SrcFld` — This can be a field name defined by the `SrcTag` "record", or a reference that will evaluate to such a field name. If the `SrcTag` is an ARS record, the `SrcFld` may also be specified as a field id.

When a `SrcFld` is specified, the field and value specific assignments are set.

Note that if you are only interested in the value when the `SrcTag` and **SrcFld** are themselves references, then the **@val** assignment will return that single value.

The following table lists the assignments made to the Tag.

Name	Type	Meaning	Initial Value
DefinedTag	bool	Tag is defined	0
DefinedField	bool	Field is defined	0
Type	Undefined, ARS, File, SQL, String	Tag type	Undefined
TypeSchema	Regular, Join, View, Dialog, Vendor	ARS Schema Type	""
Join	bool	TypeSchema is Join	0
Join1	string	Join schema 1	""
Join2	string	Join schema 2	""
View	bool	TypeSchema is View	0
ViewName	string	the database view name	""
ViewKey	string	the database key field (request id)	""
Vendor	bool	TypeSchema is Vendor	""
VendorName	string	Vendor name identifies the plugin supplying the table	""
VendorTable	string	Vendor Table is selected when defining the table to ARS	""
KeyZeroFill	bool	Set false only if the schema has defined max length of field '1' as 0	1

KeyLen	integer	Length of the request id field ('1'), almost always 15	15
KeyPfx	string	The initial value of the request id field. Acts as a prefix to an integer.	""
KeyPfxLen	integer	Length of the request id field's initial value or prefix	0
ArchEnable	bool	Archiving enabled	0
ArchType	string	One of None, Form, Delete, Form&Delete, XML, ARX	None
ArchDelete	bool	The Archive Type has the Delete flag on	None
ArchName	string	The Archive Form Name	""
ArchNoAttach	bool	The "No Attachments" option	0
ArchNoDiary	bool	The "No Diary Fields" option	0
FieldTypeInt	integer	ARS field type integer	""
FieldType	Integer, Real, Char, Diary, Enum, Time, Bitmask, Bytes, Decimal, Attach, Currency, Date, Time_of_day, Join, Trim, Control, Table, Column, Page, Page_holder, Attach_pool, Ulong, Coords, View, Display	ARS field type as a string	""
FieldId	integer	ARS field ID	0
FieldName	string	Field name	""
FieldLabel	string	ARS default field label	""
FieldDisplayOnly	bool	ARS field is DisplayOnly	0
FieldRequired	bool	ARS field is required	0
FieldMaxLength	integer	ARS maximum field length	0
Value	string	The field value	""
ValueLength	integer	The length of the value	0

Doubled Reference Values — assigning a single value using references for a Tag and Field:

When you want only the value given by a double reference – that is when the Tag and Field are themselves references – then the **`@val`** is simpler and faster than the **`@info`** command above.

```
@Cmd    =  Reference,  Tag,   Name,  @val,   SrcTag,  SrcFld
```

`@val` This is a keyword and must be coded exactly as shown. This command assigns the value of a dereferenced Tag and Field to a single string.

`SrcTag` This can be any string expression or a constant. It must evaluate to a known Tag.

`SrcFld` This can be any string expression or a constant. It must evaluate to a known Field within the Tag given. If the Tag is an ARS record, the field can be a field's name or ID.

Both the Tag and Field may be references which must evaluate to an existing or loaded Tag and a field defined in that Tag.

This allows you to hold field references in variables and do use hashes and arrays.

Note that the **`@info`** reference command also retrieves the value and can be used anywhere that this command can be used.

In the next example, assuming the Tag "**`Src`**" is a loaded ARS record with a field of Status, the next Reference command will assign that Status string to a variable:

```
@Cmd    =  Reference,  V,   Tag,  "Src"
@Cmd    =  Reference,  V,   Fld,  "Status"
@Cmd    =  Reference,  V,   Sta,  @val,   $V, Tag$,  $V, Fld$
```

If the field is an attachment, only the attachment's file name value is set. The actual attachment cannot be assigned using this facility.

Formatting Values — assigning a single value by transforming with a format

You can use a format assignment to transform a value according to a "format string". Format Strings" are used in field declaration to interpret SQL or CSV columns or as an output transformation for CSV columns. See page 148, "Field Sections" in "Script Reference" for detailed information on format strings.

There are two "forms" of a format reference assignment command:

```
@Cmd    =  Reference,  Tag,   Name,  @fmt,      Val,  Fmt
@Cmd    =  Reference,  Tag,   Name,  @fmtout,   Val,  Fmt
```

`@fmt`
`@fmtout`
These are keywords and must be coded exactly as shown.
`@fmtout` causes the transform to behave as though the field were being prepared for an output CSV file.
`@fmt` causes the opposite; that is the field is being interpreted for internal use.

This can affect dates for example. **`@fmt`** of a date always results in an internal date and time stamp which can be used in ARS assignments and **`@date`** reference assignments. **`@fmtout`**, on the other hand, can yield a wide variety of strings for a date.

`Val`
This is the source value. It can be a string reference. Quote this value if needed.

`Fmt`
This is the format specification. It can be a string reference. Quote this value if needed.

Consider that you have an SQL column containing a Remedy time stamp value. You add a number of seconds to it and want to convert it to a date to be assigned to another Remedy field. This code fragment will do that:

```
@Cmd    =  Ref,  V,  NewDate,  @eval  $Sql, cDate$ - $V, Secs$
@Cmd    =  Ref,  V,  NewDate,  @fmt,  $V, NewDate$, "Date: epoch;"
```

Say you want to substitute the Remedy Dropdown list word for an integer, you could do the following:

```
@Cmd    =  Ref,  V,  DropName,  @fmt  $Sql, DopVal$        &
           "Subst /0/New/                                  &
           "Subst /1/Open/                                 &
           "Subst /2/etc/"
```

Date Information — assigning date information

The **`@date`** command assigns a specific set of fields describing the single date reference in local time.

```
@Cmd    =  Reference,  Tag,   @date,  date
```

`@date`
`@date` is used to give information such as the day of week and the "epoch" value of any date into a specific set of fields into the specified Tag (which can be a reference).

`date`
this can be any Meta-Update recognized date. This can be a reference that evaluates to such a date.

Meta-Update dates are of the form "1999/12/31 23:59:59" When a date is read from a Remedy Time Stamp or Date field, Meta-Update converts that date into the above format. Dates from files or SQL fields may be converted by value interpretation.

The following table lists the assignments made to the Tag.

Name	Type	Meaning		Initial Value
DateType	string	**Null**	**$NULL$** or ""	Null
		Date	valid date	
		Error	invalid or unrecognized date	
IsDaylight	bool	1 if the date is in the daylight savings time zone, else 0		0
epoch	int	The Remedy epoch date in number of seconds past 00:00 at Jan 1, 1970 UT		0
year	int	Year		0
month	int	Month number 1..12		0
day	int	Day of month		0
daywk	int	Day of week with Sunday as 0		0
hour	int	Hour 0..23		0
min	int	Minute 0..59		0
sec	int	Second 0..59		0

The following script fragment will assign a new variable – **Varc, Dte** – as exactly one year back from the current date.

```
@Cmd    =  Ref,    Vnow,   @date,  $TIMESTAMP$
@Cmd    =  Ref,    Varc,   yr,     @eval                  &
                   $Vnow, year$ - 1
@Cmd    =  Ref,    Varc,   Dte,                           &
                   $Varc, year$/$Vnow, month$/$Vnow, day$  &
                   $Vnow, hour$:$Vnow, min$:$Vnow, sec$
```

The following will do the same but with the date being *approximately* one year ago:

```
@Cmd    =  Ref,    Vnow,   @date,  $TIMESTAMP$
@Cmd    =  Ref,    Varc,   epoch,   @eval                 &
                   $Vnow,  epoch$ - 365.25 * 24 * 60 * 60
@Cmd    =  Ref,    Varc,   Dte-flds,   @regex,            &
                   /(.*)/, $Varc, epoch$
[Dte-flds]
Dte     =  $       Date: epoch
```

Conditional Value Assignments to a Tag reference:

```
@Cmd    =  Reference,  Tag,   Name,  @LookUp, ..Sec,   Value
@Cmd    =  Reference,  Tag,   Name,  @if( ..)
```

If the condition coded evaluate to false, no assignment is made. If the variable is then referenced, an error will be thrown. You may get around this by assigning a default value first as in the following example.

```
@Cmd    =  Reference,  Tag,   Name,  "Initial Value"
@Cmd    =  Reference,  Tag,   Name,  @if( ..)
```

Arithmetic expressions:

```
@Cmd     =   Reference,  Tag,    Name,   @eval,   [real,]   exp
```

`@eval` This is a keyword and must be coded exactly as shown. This command assigns the value of an arithmetic expression to the Named variable

`real` This is a keyword and must be coded exactly as shown. This causes the expression to be evaluated as a floating point real number.

`Value` This is an arithmetic expression. It can contain references, parentheses, arithmetic operators, and basic functions. Normal arithmetic precedence rules apply and can be changed by the use of parentheses.

The result of the expression is an integer if the “real” keyword is not coded. The interim processing of the expression is done with real numbers and the value is rounded up to the nearest integer. With the real keyword there is no rounding. The floor function may be used to implement rounding.

References that result in the value $NULL$ are treated as 0.

Please see ***Using Arithmetic Expressions*** below for details on the arithmetic operators and functions supported.

Equivalent Tags – Assigning a Tag as another Tag

```
@Cmd     =   Reference,  Tag,    Name,   @equ
```

`@equ` This is a keyword and must be coded exactly as shown.

This assigns an equivalent Tag so that fields and references are entirely equivalent using either tag. This is useful when assignment sections are included and made to act on different records.

Name is interpreted as a previously defined Reference Tag.

Server Processes – Assigning results of ARS Server Run Process

```
@Cmd     =   Reference,  Tag,    Name,   @guid    [ prefix ]
```

`@guid` This is a keyword and must be coded exactly as shown.

This causes this assignment to assign an ARS GUID as per the special run process `Application-Generate-GUID [<GUID prefix>]` executed on the target server. A zero, one, or two character prefix may be passed as an argument. A one character prefix is suffixed with an underscore. No prefix results in ID being used.

```
@Cmd    =  Reference,  Tag,   Name,   @exec   process  [ arguments ]
```

`@exec` This is a keyword and must be coded exactly as shown.

This causes this assignment to assign the results of the special run process coded on the statement, along with any parameters required for that special process.

For example

```
@Cmd = Ref, X, Guid, @exec, Application-Generate-GUID AA
```

would be equivalent to

```
@Cmd = Ref, X, Guid, @guid, AA
```

Similarly, calls can be made to any of the Special Run Processes available and listed in the table at the end of the BMC: ARS System 7.x Workflow Objects documents.

Regular Expressions – Assigning match and extracts to variables

```
@Cmd    =  Reference,  Tag,   Name,  @regex   regex,  Value
```

`@regex` This is a keyword and must be coded exactly as shown.

This causes this assignment to assign several implied named strings to the specified Tag.

A Perl compatible regular expression is specified in regex and the supplied, de-referenced value is tested against this regular expression.

The regular expression itself may also contain references. This is useful when the regular expression is dynamic, depending on other script or record variables.

The Tag's field **`@rc`** is set to "1" when the pattern is matched or to "0" when the pattern is not matched. "No match" does not cause an error. You should test **`$Tag, @rc$`** before relying on the substrings extracted from the pattern.

All pattern specified output strings are extracted and set in the Tag under a name represented by the extracted string's index (starting at 1).

`Name` If **`Name`** is specified and not **`@na`**, a field section is processed and extracted values are transformed according to the field section's format specifications. Additionally, variables with the field section's field names are set in the Tag.

A field "**`@match`**" can be specified as the first field. If so specified this contains the whole string that matched the complete expression.

`regex` This is a Perl Compatible regular expression. It will be used to match against the value given and extract any matching substrings from that value.

In accordance with the Perl convention, the first character is treated as a delimiter and the expression is considered complete at the next such character.

See Using regular expressions below for more information.

`Value` This is string that the regular expression matches and extracts substrings from.

Assigning values to an ARS record:

```
@Cmd     =  Reference,  Tag,   Name,  @ars,                          &
            [@SvrTag,]  Schema  [,  @init]
```

`@ars` This is a keyword and must be coded exactly as shown. This command is used to make assignments to a single ARS record as identified by the Tag.

If Tag has not been encountered before, or if the optional **`@init`** is coded, it is allocated with no field values.

Tag The name that this record's fields will be referenced by. This cannot be used directly in an Update assignment but can be used in normal references and in the Copy assignment command.

`Name` This is an assignment section that will be applied with this record as a target. If this section causes an error or issues an Abort, this assignment will also be aborted or be in error.

Name can be specified as **`@na`** which causes no assignment section to be processed.

Name can be a string reference.

`@SvrTag` This is an optional ReadServer Tag. As with all Read Servers, the @ is required. It specifies the server that the schema is loaded from.

`Schema` This is the name of the ARS Schema or form that the record will belong to. This can be specified as a constant or a string reference.

`@init` This is an optional keyword. If coded, the record will be initialised to the empty record. That is a record containing no field value pairs and having had no assignments.

If this keyword is not specified, this section's assignments add to the record. Several of these commands can be used to accumulate values in the record.

Client process' stdout and stderr files – Assign to a Tag:

```
@Cmd    =  Reference,  Tag,   @spawn,  process  [ arguments ]
```

`@spawn` This is a keyword and must be coded exactly as shown. This command is used to make assignments to three specific string names on the specified Tag

rc	the spawned process' returned integer
stderr	the stderr output (console errors) of the command; if the process succeeded this is generally empty: ""
stdout	the stdout output (console) of the command

The process must be on the path when Meta-Update starts.

The process cannot have redirection operators for stdout and stderr. These are appended by Meta-Update. These are temporary files that will be automatically deleted after the command runs.

If desired, the placement of Meta-Update's stdout and stderr redirects may be controlled by use of the **$redir$** string. If missing from the text to spawn, the redirects are added to the end of the command text.

If there are multiple lines, they are concatenated into a single string containing the line ends. You may use a `Loop=` if needed to process these lines individually (using a line feed as the delimiter).

Alternatively, you may use a normal Spawn command and then process the files with a `File=`.

This example, run on Windows with Cygwin installed, will extract the Windows User Id to produce an information level message:

```
@Cmd    =  Reference,  V,   @spawn,                                  &
           set | grep USERNAME  | cut -d "=" -f 2
@Cmd    = @if("$V, rc$" == 0)
  @Cmd    =  Msg, I, User is $V, stdout$
@Cmd    =  else
  @Cmd    =  Msg, W, Spawn for Windows User returned $V, rc$
@Cmd    =  endif
```

This example will set the contents of a file into a field:

```
@Cmd    = @if("$CTL, OS$" == "Windows")
  @Cmd  =    Reference, V, @spawn,  type  $Arg, filename$
@Cmd    =  else
  @Cmd  =    Reference, V, @spawn,  cat   $Arg, filename$
@Cmd    =  endif

Field   =  V,  stdout
```

Using Regular Expressions

Regular expressions may be used to match and extract (split) values.

This is an example of a script that does no ARS updates but simply splits the specified string around the last " / " and trims and leading and trailing spaces from both parts:

```
[DoSplit]
PrmReq      =  1, Usage  $CTL, script-f$ DoSplit -p subj »
ArgNm       =  subj
AssignInit  =  asg-Split

[asg-Split]
@Cmd   = Ref,  X,  regex-parts,  @regex,                  &
                  '(.*) / (.*)',                          &
                  "$Arg, subj$"
@Cmd   = @if("$X, @rc$" == "1")
  @Cmd   = Msg, I, "matched: Src:    $Arg, subj$"
  @Cmd   = Msg, I, "matched: Part 1: $X, part 1$"
  @Cmd   = Msg, I, "matched: Part 2: $X, part 2$"
@Cmd   = else
  @Cmd   = Msg, W, "no match: $Arg, subj$"
@Cmd   = endif

[regex-parts]
part 1    = $  Trim both
part 2    = $  Trim both"
```

When run, the following output is generated

```
SthMupd.exe  BBB-Asg-regex-010.ini Do -p "Model 132 / 42 / Manu"
[DoSplit] Msg: matched: Src:    Model 132 / 42 / Manu
[DoSplit] Msg: matched: Part 1: Model 132 / 42
[DoSplit] Msg: matched: Part 2: Manu
```

Meta-Update's regular expression handling is through the PCRE libraries. PCRE is the Perl Compatible Regular Expression implementation available as a GNU project.

PCRE can modify the regular expression behaviour by including options between "(?" and ")". By prefixing an option letter by a hyphen, that option is turned off in the following pattern part.

The option letters are:

Letter	Option	Meaning
i	PCRE_CASELESS	If this modifier is set, letters in the pattern match both upper and lower case letters.
m	PCRE_MULTILINE	PCRE treats the subject string as a single "line" of characters (even if it contains several newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline (unless D modifier is set). When this modifier is set, the "start of line" and "end of line" constructs match immediately following or immediately before any newline in the subject string, respectively, as well as at the very start and end. If there are no newlines in a subject string, or no occurrences of ^ or $ in a pattern, setting this modifier has no effect.

s	PCRE_DOTALL	If this modifier is set, a dot metacharacter in the pattern matches all characters, including newlines. Without it, newlines are excluded.
x	PCRE_EXTENDED	If this modifier is set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class, and characters between an unescaped # outside a character class and the next newline character, inclusive, are also ignored. This is equivalent to Perl's /x modifier, and makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (?(which introduces a conditional subpattern.
U	PCRE_UNGREEDY	This modifier inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by ?. It is not compatible with Perl. It can also be set by a (?U) modifier setting within the pattern or by a question mark behind a quantifier (e.g. .*?).
X	PCRE_EXTRA	This modifier turns on additional functionality of PCRE that is incompatible with Perl. Any backslash in a pattern that is followed by a letter that has no special meaning causes an error, thus reserving these combinations for future expansion. By default, as in Perl, a backslash followed by a letter with no special meaning is treated as a literal.

There are some differences in regular expression handling among all regular expression engines. For complete information on regular expressions, and specifically, the regular expressions implemented by PCRE, please refer to the PCRE or regex man pages available on the web.

Using Arithmetic Expressions

Arithmetic expressions can be assigned to string variables as either integers or real numbers. Here are a few examples:

```
@Cmd    =  Ref,    MyVars,   Ctr,  @eval,  $MyVars, Ctr$ + 1
@Cmd    =  Ref,    MyVars,  Area,  @eval,  pi * ($CiXmit, range$ ^ 2)
```

The following unary operator is supported:

- unary minus

The following binary operators are supported:

***** multiplication
/ division
^ exponentiation
+ addition
- subtraction

The usual arithmetic rules of precedence apply. You can change the order of evaluation by using parentheses.

All arithmetic functions are implemented with the GNU matheval library which comes with support for some named mathematical constants and basic functions. While of improbable use in a Remedy application, these are documented here for completeness.

The following named constants are available.

e	e	2.718282
log2e	log2(e)	1.442695
log10e	log10(e)	0.434294
ln2	ln(2)	0.693147
ln10	ln(10)	2.302585
pi	pi	3.141593
pi_2	pi / 2	1.570796
pi_4	pi / 4	0.785398
1_pi	1 / pi	0.318310
2_pi	2 / pi	0.636620
2_sqrtpi	2 / sqrt(pi)	1.128379
sqrt2	sqrt(2)	1.414214
sqrt1_2	sqrt(1/2)	0.707107

The following elementary functions are available:

abs(x)	absolute value of x
sqrt(x)	square root if x
rand(x)	a random number between 0 and x
floor(x)	returns nearest integer value for x
ciel(x)	returns smallest integer value that is greater than or equal to x.
exp(x)	exponential of x
log(x)	logarithm of x
sin(x)	sine of x where x is in radians
asin(x)	inverse sine of x
cos(x)	cosine of x
acos(x)	inverse cosine of x
tan(x)	tangent of x
atan(x)	inverse tangent of x
cot(x)	cotangent of x
acot(x)	inverse cotangent of x
sec(x)	secant of x equivalent to 1/cos(x)
asec(x)	inverse secant of x
csc(x)	cosecant of x
acsc(x)	inverse cosecant of x
sinh(x)	hyperbolic sine of x
asinh(x)	inverse hyperbolic sine of x
cosh(x)	hyperbolic cosine of x
acosh(x)	inverse hyperbolic cosine of x
tanh(x)	hyperbolic tangent of x
atanh(x)	inverse hyperbolic tangent of x
coth(x)	hyperbolic cotangent of x

acoth(x)	inverse hyperbolic cotangent of x
sech(x)	hyperbolic secant of x
asech(x)	inverse hyperbolic secant of x
csch(x)	hyperbolic cosecant of x
acsch(x)	inverse hyperbolic cosecant of x

Note that 1 degree = 0.0174532925 radians.

The **`rand()`** function uses the standard OS implementations of rand(). As such, the limitations associated with the standard random number generators are inherent in the Meta-Update generator.

The function is seeded with the current time at the start of the Meta-Update job. This is true even if the random number function is not used in the script. The first 100 random numbers are discarded as part of the seeding process.

Seeding, and the discarding of the first 100 results, is automatic but can be inhibited with the `RandSeed = No` directive in the `[Main]` section.

If seeding is inhibited, each run of Meta-Update will produce the same sequence of random numbers.

Set Schema Command

The Set Schema command allows you to alter some form parameters. Currently only the Archive settings for a form may be set.

The Set Schema Command alters the definition of the specified form. This is not a data operation.

This requires Admin privileges and should be used with caution.

```
@Cmd    =   Set Schema Archive      Schema-Name        SrcTag
```

`Set`	This must be coded exactly as shown and indicates a Set command.
`Schema`	Must be coded as shown and indicates that a form property will be changed.
`Archive`	Must be coded as shown and indicates that a form's archive property will be changed.
`Schema-Name`	This is a reference or constant with the name of the ARS form. Any form name with spaces or special characters should be enclosed in quote marks.
`SrcTag`	This is a reference to a string Tag that contains a minimum number of specific fields and appropriate function for the Set operation being performed.

For the Archive settings, the following fields may be set:

Examples:

```
[Do]
AssignInit  = asg-Arch,  asg-None

[asg-Arch]
@Cmd    =   Ref,  Arch,     ArchName,    "HPD:Help Desk-ARC"
@Cmd    =   Ref,  Arch,     ArchType,    "Form"
@Cmd    =   Ref,  Arch,     ArchDelete,  1
@Cmd    =   Ref,  Arch,     ArchEnable,  "true"
@Cmd    =   Set,  Schema,  "HPD:Help Desk"     Arch

[asg-None]
@Cmd    =   Ref,  Arch,     ArchName,    ""
@Cmd    =   Ref,  Arch,     ArchType,    "None"
@Cmd    =   Ref,  Arch,     ArchEnable,  "false"
@Cmd    =   Set,  Schema,  "HPD:Help Desk"     Arch
```

In the above script, the initial assignment section **`[asg-Arch]`** will cause Remedy to set the connection between the main and archive forms of **`"HPD:Help Desk"`** and **`"HPD:Help Desk-ARC"`**, creating the archive form if it doesn't already exist.

Then, the second initial assignment section, **`[asg-None]`** will reset the Archive Properties of **`"HPD:Help Desk"`** to have no archiving defined. This will sever the connection between the two forms, **`"HPD:Help Desk"`** and **`"HPD:Help Desk-ARC"`** , but will not delete the archive form. The archive form will now be considered a regular form.

Trace Command

The Trace command allows push, pop, and change the trace settings, when the script is run with tracing. See Running Meta-Update, The Command Line for tracing scripts.

The trace command is generally used while debugging scripts by inhibiting or reducing tracing in already debugged sections and then selectively tracing other sections.

```
@Cmd    =   Trace Push
@Cmd    =   Trace Trc-Lvl
@Cmd    =   Trace Pop
```

`Push`	This is used to save the current trace levels.
`Pop`	Resumes the trace levels at the time of the matching Push. Meta-Update will issue a Warning when a Pop is used without a previous Push.
`Trc-Lvl`	A Trace Level setting. See Running Meta-Update, Tracing for more information.

Note: Trace commands within a script will be ignored when run with the minus minus d switch: **--d**. See Running Meta-Update, Tracing for more information.

LookUp Sections

Overview

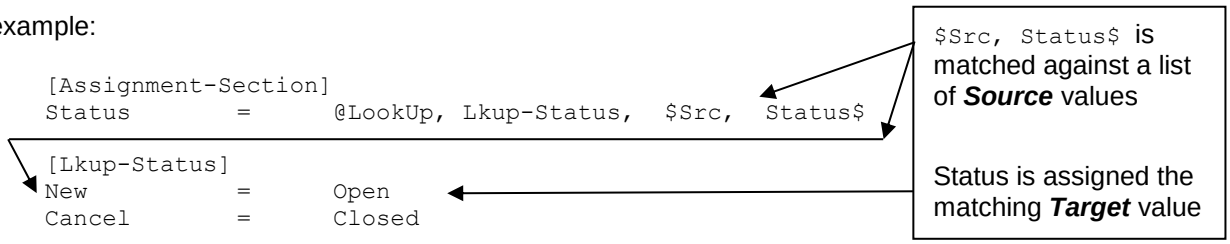
A LookUp section is used in the @LookUp assignments translate values and load records using lists, files, ARS and SQL queries.

A LookUp section can be used in field or reference assignments. Different @LookUp assignments may refer to a single LookUp section.

```
Field  =  @LookUp,   LookUp-Section,    Source-Value

@Cmd   =  Ref,  V,  New-Status,          &
          @LookUp,   LookUp-Section,    Source-Value
```

The @LookUp assignment refers to a LookUp section and passes that look up section a source string. That source string is then matched against the source – left - side of that list of pairs, and, if found, the corresponding target – right - side of that pair of strings is returned:

An example:



If the source string, as specified by `$Src, Status$`, is **"New"** then **"Open"** is returned and will be assigned to the Status field or the script variable V, New-Status.

A LookUp can also be used to load ARS records or SQL rows by issuing queries the Remedy. These records are then used to create the returned target string and are also available to the script.

LookUp Types

A LookUp can translate a source string into a target string through one or more of these sources:

- A list of value pairs in the LookUp section

```
LookUp Val    =      Val
LookUp Val 2  =      Val 2
   . . .
```

- An external file such as a CSV or any columnar file.

```
Fld-1|Fld-2|Fld-3|Fld-4
Val-1-1|Val-1-2|Val-1-3|Val-1-4
Val-2-1|Val-2-2|Val-2-3|Val-2-4
Val-3-1|Val-3-2|Val-3-3|Val-3-4
   . . .
```

  The first time a LookUp that uses an external file is used, the file is read and a list of source and target pairs is generated from the values in the file. When the same

LookUp is used again, the list that was read is used again and no more file reads happen.

- An ARS Query or SQL Query.

  The selected record, if found, is Loaded. A result string is made from the fields of that record. The Loaded record can also be used by the rest of the script. If the Query returns no results, it is still possible for the LookUp to succeed through another source listed above.

  These records may optionally be cached so that if the record is found once for a source string, and that same source string is applied to the same LookUp section, the same record will be returned without executing the Query.

  Caching of records is not on unless specified. The default behaviour is to not cache records. With caching in a LookUp section, the time required to access the server can be eliminated significantly reducing the time required for a data operation.

A special reference is set to indicate the results of the LookUp. This reference can be queried to determine that there is a loaded record available for use.

An SQL statement opens up the power of SQL functions to translate a value.

Any and all of the above sources may be used in a LookUp section.

Automatic Tags

The LookUp section sets automatic CTL variables each time it is used in a LookUp assignment. Two variables are automatically set. These are used to specify the LookUp source string within the LookUp section itself, and to specify where the return string was found.

Each LookUp assignment sets these same variables. If these variables are needed by the script, they should be saved in script variables.

CTL	**LookUp_Src**	The source string from the LookUp assignment.
CTL	**LookUp**	Set when a LookUp is successful to contain one of these values:
	Default	The string was not found; the default was returned.
	List	Found in the internal List
	File	Found in the external File
	Query	Found in and loaded an ARS record.
	QuerySql.	Found in and loaded an SQL row.

Keywords

These keywords have special meaning in a LookUp section. All other keywords become source and target strings of an internal LookUp list.

Default	Optional. The value to return if the LookUp string is not found. **$CTL, LookUp_Src$** may be used to return the LookUp string itself. Default $NULL$
NoMatch	Optional. Specifies the message level when the LookUp is not found as well as the action to be taken. Default: E, Error
Order	Optional. Specifies the Order of LookUp lists to be searched. Use any of the words in the default order below arranged in the order that the LookUp will be processed. Default: List, File, Query, QuerySql
File	Optional. Specifies that an external CSV file is to be used to load a table of LookUp value pairs. A **File=** specifies a text file including any Meta-Update references that is copied into the target named string or pattern file "record". `File = File-Tag, File-Section, $Arg, Filename$` See the **File=** statement in the Script Reference above.
FileSource	Required when **File=** is used. Specifies the string to be built as the Source (LookUp) while loading the file. Use **Tag, fld** to specify fields of the file.
FileTarget	Required when **File=** is used. Specifies the string to be built as the Target (returned value) while loading the file. Use **Tag, fld** to specify fields of the file.
FileIf	Optional. Only used with **File=**. Specifies a condition that must be satisfied for a record to be included. The file record's fields are referenced through the Tag specified in the **File=** statement. `FileIf = @if("$File-Tag, fld-1$" == "Active")`

Query	Optional. Specifies that an ARS Query will be loaded and used to derive the returned string, if matched.
	
	A **Query=** specifies a query that should return exactly one row. The **$CTL, LookUp_Src$** reference can be used in the Query.
	
	Unlike a **LoadQ=** a LookUp query can return zero, one, or more than one record. Other keywords control what to do when the number of records returned is not exactly one.
	
	`Query = Qry-Tag, QRY:Schema, &`
	`'fld-test' = "$CTL, LookUp_Src$" &`
	`'Status' = "Active"`

QueryTarget	Required when **Query=** is used.
	
	Specifies the string to be built as the LookUp return value when the LookUp query matches a row. Should use references within the loaded query record to create the target string.
	
	`QueryTarget = $CTL, LookUp_Src$ - &`
	`$Qry-Tag, fld1$lename$`

QueryMulti	Optional when **Query=** used. Default is "**Error**" Specifies the action to take when multiple records are returned by the ARS query when the LookUp is done. Values are "**Error**" and "**First**". If "**First**" is selected, the first record that matches is loaded into the Tag and the LookUp return string is made fusing that loaded record.

QuerySql	Optional. Specifies that an ARS SQL Query will be loaded and used to derive the returned string, if matched.
	
	A **QuerySql=** specifies an SQL query that should return exactly one row. The **$CTL, LookUp_Src$** reference can be used in the Query.
	
	The full features of the **QuerySql=** statement are available. This includes the field value interpretations and transformations..
	
	`QuerySql = Qry-Dwl-Tag, @na, &`
	`Select fld-val from QRY_Schema &`
	`where fld-test' = &`
	`'$CTL, LookUp_Src$' and &`
	`Status = 2`

QuerySqlTarget	Required when **QuerySql=** is used. Specifies the string to be built as the LookUp return value when the LookUp query matches a row. Should use references within the loaded query record to create the target string. SQL columns are numbered starting at 1, or field names can be used if defined on the **QuerySql=** statement. `QueryTarget = $CTL, LookUp_Src$ - & $Qry-Tag, fld1$lename$`
QuerySqlMulti	Optional when **QuerySql=** used. Default is "**Error**" Specifies the action to take when multiple records are returned by the ARS SQL query when the LookUp is done. Values are "**Error**" and "**First**". If "**First**" is selected, the first record that matches is loaded into the Tag and the LookUp return string is made fusing that loaded record.
Cache	Optional when **Query=** or **QuerySql=** is used. Default is "**off**" Specifies the keyword "**off**" or a number of records to cache. Zero indicates an unlimited cache. If **Cache=** is specified then any records that match a source string are saved in memory and set as though they have been retrieved again by the Query See Caching LookUp Records below for more information on the LookUp cache.

The simplest @LookUp may include:

```
[LookUp Section]
Default       =      $CTL, LookUp_Src$
NoMatch       =      { I, D, W, E } [ , { Default, Skip, Error } ]

LookUp Val    =      Return Val
LookUp Val 2  =      Return Val 2
   . . .
```

Default Specifies a string reference to be used as the value returned when an exact match is not made. The Default value is $NULL$. The special symbol **$CTL, LookUp_Src$** may be used. It refers to the value passed to the LookUp section. It is the value being looked up.

NoMatch Specifies the actions to be done when the source value is not matched. Default is E, Error

The first value is the type of message that will be traced.

I	Information	Always logged
D	Debug	Only logged if –d was specified.
W	Warning	Always logged.

E	Error	Always logged.

The second value indicates the action to be done if the source value is not matched.

Default	The default value coded is taken.
Skip	No assignment is made to this field.
Error	An error is returned and this operation is aborted.

Val Specifies the value reference that will be returned as a result of this LookUp when the source value being looked up matches the associated LookUp value precisely.

LookUp Val Specifies a string constant that will be matched against the passed source string reference. When this constant is matched exactly, its associated Val will be the result of the LookUp command.

Examples"

```
[xlt-AsigneeLogin]
Default       =      $CTL, LookUp_Src$
NoMatch       =      D, Default
#                          Ms J Blow changed names
jblow         =      jsmith
#.............................These persons don't exist in the
#                          new system.  Make the Assignee $NULL$
jdoe          =      $NULL$
```

Using Files

LookUp sections can also use external files for its lists of string pairs.

To use external CSV files use add the **`File=`** keyword to the LookUp section.

When you use the **`File=`** keyword, all other file related keywords are also required. A list of value pairs within the LookUp section will override, or be overridden by, the list from the external file. The **`Order=`** keyword controls the order in which the lists are searched.

```
[LookUp Section]
Default       =      xxx
NoMatch       =      { I, D, W, E }
                     [ , { Default, Skip, Error } ]

LookUp Val    =      Val
LookUp Val 2  =      Val 2
   . . .

File          =      Tag, LookUp-File, $Arg, File Name$
FileSource    =      $Tag, Fld1-Src1$|$Tag, Fld3-Src2$
FileTarget    =      $Tag, Fld3-Tgt1$|$Tag, Fld4-Tgt2$
Order         =      File, List
FileIf        =      @if ("$Tag, Fld1-Src1$" ~= "Test")

[LookUp-File]
Type          =      Csv
Format        =      Excel
Fields        =      LookUp-File-Fields

[LookUp-File-Fields]
Fld1-Src1     =      $      Trim both
Fld2-Tgt1     =      $
Fld3-Src2     =      $
Fld4-Tgt2     =      $
```

If a list is coded in the section, the **`Order=`** keyword controls the list search sequence.

String pairs are mapped to the fields of the file.

The **`If=`** puts a filter on the file's records. Only records that match the condition are loaded

`File`

Specifies that an external columnar file will be used to create the LookUp table.

If coded, all other keywords below are required.

The **`File=`** keyword and syntax as well as the file definition in the specified File section is exactly as described in File Sections in the Command Reference part of this document above.

The **`Tag`** coded in the **`File=`** is only used during the initial load of the file when the LookUp section is first used. Any other references to that **`Tag`** will fail.

`FileSource`

This allows you to specify how the source string is to be created from the fields in the file. It is evaluated once only when a LookUp section is first used.

The **`FileSource=`** string specifies file fields and other constants and is used to build the table of source strings that will be used in the LookUp.

Any Meta-Update references are evaluated once when the file is loaded.

As each record is loaded, it is placed into the **`File=`** Tag specified. You can then use that Tag in the string reference.

This simple example uses a single column of the file as the list of source strings:

```
FileSource    =      $Tag, Fld1-Src1$
```

In this example, the source strings are made up of two file columns and a separator.

```
FileSource    =      $Tag, Fld1-Src1$ | $Tag, Fld2-Src2$
```

`FileTarget` Similar to the **`FileSource=`** value, use this to specify how to build the returned LookUp string from the fields in the file. This setting is evaluated once only when the file is loaded and the LookUp section is first referenced

In this example, the source and target strings are each made of two file columns and a separator.

```
FileSource    =      $Tag, Fld-Src1$ | $Tag, Fld-Src2$
FileTarget    =      $Tag, Fld-Tgt1$ | $Tag, Fld-Tgt2$
```

`FileIf` This specifies a condition that, if true, causes the record to be inserted into the LookUp tables. If false, the record is ignored.

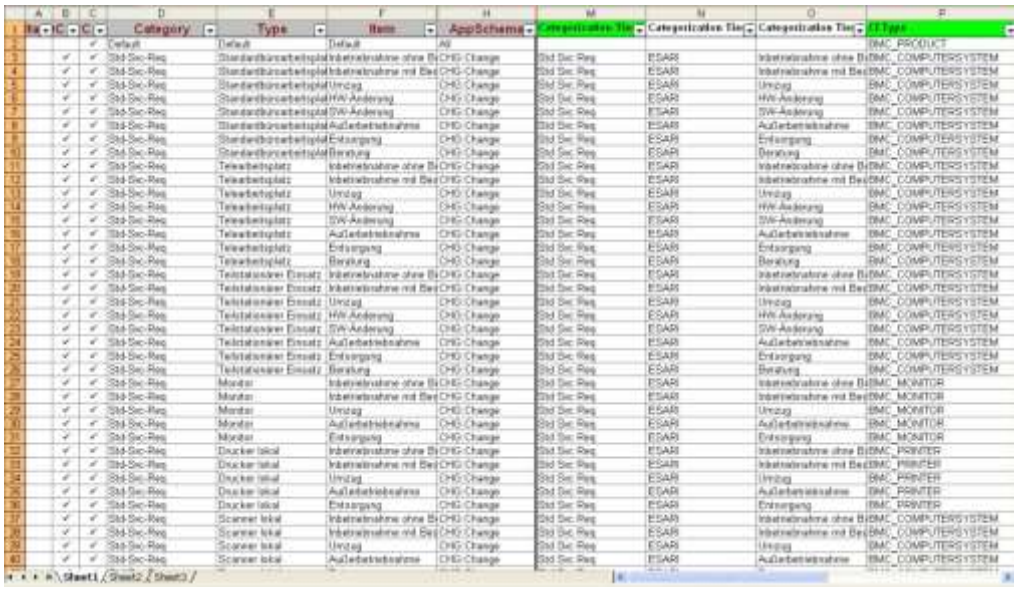
The condition may use the **`File=`** Tag for the file record's reference.

In the CSV below, we may want to exclude all records that are not for the CHG:Change application:

```
FileIf        =      @if ("$Tag, AppSchema$"  ==       &
                                    "CHG:Change")
```

This more complete example is described below:

This image is of a sample CSV from an ITSM 6 to ITSM 7 Migration script describing all CTI, Product, Model conversions. Different slices of the same file were used in different LookUp sections.



	A	B	C	D	E	F	H	M	N	O	P
1	Ite	IC	C	Category	Type	Item	AppSchema	Categorization Tie	Categorization Tie	Categorization Tie	CITypo
2				Default	Default	Default	All				BMC_PRODUCT
3				Std-Svc-Req	Standardbüroarbeitsplat	Inbetriebnahme ohne Be	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme ohne Be	BMC_COMPUTERSYSTEM
4				Std-Svc-Req	Standardbüroarbeitsplat	Inbetriebnahme mit Ber	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme mit Ber	BMC_COMPUTERSYSTEM
5				Std-Svc-Req	Standardbüroarbeitsplat	Umzug	CHG:Change	Std Svc Req	ESAR	Umzug	BMC_COMPUTERSYSTEM
6				Std-Svc-Req	Standardbüroarbeitsplat	HW-Änderung	CHG:Change	Std Svc Req	ESAR	HW-Änderung	BMC_COMPUTERSYSTEM
7				Std-Svc-Req	Standardbüroarbeitsplat	SW-Änderung	CHG:Change	Std Svc Req	ESAR	SW-Änderung	BMC_COMPUTERSYSTEM
8				Std-Svc-Req	Standardbüroarbeitsplat	Außerbetriebnahme	CHG:Change	Std Svc Req	ESAR	Außerbetriebnahme	BMC_COMPUTERSYSTEM
9				Std-Svc-Req	Standardbüroarbeitsplat	Entsorgung	CHG:Change	Std Svc Req	ESAR	Entsorgung	BMC_COMPUTERSYSTEM
10				Std-Svc-Req	Standardbüroarbeitsplat	Beratung	CHG:Change	Std Svc Req	ESAR	Beratung	BMC_COMPUTERSYSTEM
11				Std-Svc-Req	Telearbeitsplatz	Inbetriebnahme ohne Be	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme ohne Be	BMC_COMPUTERSYSTEM
12				Std-Svc-Req	Telearbeitsplatz	Inbetriebnahme mit Ber	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme mit Ber	BMC_COMPUTERSYSTEM
13				Std-Svc-Req	Telearbeitsplatz	Umzug	CHG:Change	Std Svc Req	ESAR	Umzug	BMC_COMPUTERSYSTEM
14				Std-Svc-Req	Telearbeitsplatz	HW-Änderung	CHG:Change	Std Svc Req	ESAR	HW-Änderung	BMC_COMPUTERSYSTEM
15				Std-Svc-Req	Telearbeitsplatz	SW-Änderung	CHG:Change	Std Svc Req	ESAR	SW-Änderung	BMC_COMPUTERSYSTEM
16				Std-Svc-Req	Telearbeitsplatz	Außerbetriebnahme	CHG:Change	Std Svc Req	ESAR	Außerbetriebnahme	BMC_COMPUTERSYSTEM
17				Std-Svc-Req	Telearbeitsplatz	Entsorgung	CHG:Change	Std Svc Req	ESAR	Entsorgung	BMC_COMPUTERSYSTEM
18				Std-Svc-Req	Telearbeitsplatz	Beratung	CHG:Change	Std Svc Req	ESAR	Beratung	BMC_COMPUTERSYSTEM
19				Std-Svc-Req	Teilstationärer Einsatz	Inbetriebnahme ohne Be	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme ohne Be	BMC_COMPUTERSYSTEM
20				Std-Svc-Req	Teilstationärer Einsatz	Inbetriebnahme mit Ber	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme mit Ber	BMC_COMPUTERSYSTEM
21				Std-Svc-Req	Teilstationärer Einsatz	Umzug	CHG:Change	Std Svc Req	ESAR	Umzug	BMC_COMPUTERSYSTEM
22				Std-Svc-Req	Teilstationärer Einsatz	HW-Änderung	CHG:Change	Std Svc Req	ESAR	HW-Änderung	BMC_COMPUTERSYSTEM
23				Std-Svc-Req	Teilstationärer Einsatz	SW-Änderung	CHG:Change	Std Svc Req	ESAR	SW-Änderung	BMC_COMPUTERSYSTEM
24				Std-Svc-Req	Teilstationärer Einsatz	Außerbetriebnahme	CHG:Change	Std Svc Req	ESAR	Außerbetriebnahme	BMC_COMPUTERSYSTEM
25				Std-Svc-Req	Teilstationärer Einsatz	Entsorgung	CHG:Change	Std Svc Req	ESAR	Entsorgung	BMC_COMPUTERSYSTEM
26				Std-Svc-Req	Teilstationärer Einsatz	Beratung	CHG:Change	Std Svc Req	ESAR	Beratung	BMC_COMPUTERSYSTEM
27				Std-Svc-Req	Monitor	Inbetriebnahme ohne Be	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme ohne Be	BMC_MONITOR
28				Std-Svc-Req	Monitor	Inbetriebnahme mit Ber	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme mit Ber	BMC_MONITOR
29				Std-Svc-Req	Monitor	Umzug	CHG:Change	Std Svc Req	ESAR	Umzug	BMC_MONITOR
30				Std-Svc-Req	Monitor	Außerbetriebnahme	CHG:Change	Std Svc Req	ESAR	Außerbetriebnahme	BMC_MONITOR
31				Std-Svc-Req	Monitor	Entsorgung	CHG:Change	Std Svc Req	ESAR	Entsorgung	BMC_MONITOR
32				Std-Svc-Req	Drucker lokal	Inbetriebnahme ohne Be	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme ohne Be	BMC_PRINTER
33				Std-Svc-Req	Drucker lokal	Inbetriebnahme mit Ber	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme mit Ber	BMC_PRINTER
34				Std-Svc-Req	Drucker lokal	Umzug	CHG:Change	Std Svc Req	ESAR	Umzug	BMC_PRINTER
35				Std-Svc-Req	Drucker lokal	Außerbetriebnahme	CHG:Change	Std Svc Req	ESAR	Außerbetriebnahme	BMC_PRINTER
36				Std-Svc-Req	Drucker lokal	Entsorgung	CHG:Change	Std Svc Req	ESAR	Entsorgung	BMC_PRINTER
37				Std-Svc-Req	Scanner lokal	Inbetriebnahme ohne Be	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme ohne Be	BMC_COMPUTERSYSTEM
38				Std-Svc-Req	Scanner lokal	Inbetriebnahme mit Ber	CHG:Change	Std Svc Req	ESAR	Inbetriebnahme mit Ber	BMC_COMPUTERSYSTEM
39				Std-Svc-Req	Scanner lokal	Umzug	CHG:Change	Std Svc Req	ESAR	Umzug	BMC_COMPUTERSYSTEM
40				Std-Svc-Req	Scanner lokal	Außerbetriebnahme	CHG:Change	Std Svc Req	ESAR	Außerbetriebnahme	BMC_COMPUTERSYSTEM

This LookUp section will take as an ITSM 6 root request's AppSchema, and its Category, Type, and Item, and will return a new Categorization Tiers 1, 2, 3 from the ITSM 7 Suite.

The source string was the original record's Category, Type, Item separated by " | ". The returned string is the new Categorization Tier 1, 2, 3.

```
  @Cmd   =    @Ref,         MyVars,       CT_lkup,                  &
                @LookUp,    LkUp-CT,                                  &
                      $HpdSrc, Category$ |                            &
                      $HpdSrc, Type$ | $HpdSrc, Item$

[CT_lkup]
Default         =      $CTL, LookUp_Src$
NoMatch         =      D, Default
Override        =      List, File

|||            =      Default|Default|Default
|na|na         =      Default|Default|Default

File            =      F-CT, File-CT,                          &
                       $Arg, File Name$
FileSource      =      $F-CT, Category$|$F-CT, Type$|$F-CT, Item$
FileTarget      =      $F-CT, Categorization Tier 1$ |         &
                       $F-CT, Categorization Tier 2$ |         &
                       $F-CT, Categorization Tier 3
FileIf          =      @if ("$F-CT, AppSchema$" ==             &
                              "CHG:Change"    )

[File-CT]
Type            =      Csv
Format          =      Excel
Fields          =      File-CT-Fields

[File-CT-Fields]
Status                      =      $
OC OK                       =      $
PC OK                       =      $
Category                    =      $
Type                        =      $
Item                        =      $
AppSchema                   =      $
Asset_Class                 =      $
Categorization Tier 1       =      $
Categorization Tier 2       =      $
Categorization Tier 3       =      $
CI Type                     =      $
Product Categorization Tier 1     =      $
Product Categorization Tier 2     =      $
Product Categorization Tier 3     =      $
Product Name                =      $
Modell/Version              =      $
Manufacturer                =      $
Check?                      =      $
```

Simply augments the file with a script wide list (and overrides if in both the file and here)

Specifies the LookUp source and target string mappings to fields in the file.

Only records matching this condition are loaded.

An assignment statement that references the LookUp section based on the above file: The source string was the original record's AppSchema,Category, Type, Item separated by

```
  @Cmd   =    @Ref,         MyVars,       CT_lkup,                  &
                @LookUp,    LkUp-CT,                                  &
                    $HpdSrc, AppSchema$ | $HpdSrc, Category$ |        &
                    $HpdSrc, Type$|$HpdSrc, Item$
```

A **File=**, if coded, is read once and only once on its first use by a @LookUp reference. That initial file read creates a list of sorted value pairs according to the Source and Target keywords. Subsequent @LookUp statements using the same LookUp section simply search this cached list.

Using a Query

LookUp sections can also use ARS queries to build the translate string.

In this way, a LookUp acts like a Load that is allowed to fail. The record, if found, is made available to the rest of the script under the Query Tag specified in the `Query=` statement.

After a @LookUp assignment $CTL, LookUp$ can be used to determine if the Query was satisfied or a default or other list was used.

A string reference using fields of the LookUp record found is used to build the return string.

The Schema for the query may be a string reference. In this way, the same LookUp section may be used for different schemas.

Similar to ARS if multiple records match, you may throw an error (the default) or select the first record. You may use the optional Sort in your Query.

To use an ARS query add the **Query=** keyword to the LookUp section.

When you use the **Query=** keyword, all other file related keywords are also required. A list of value pairs within the LookUp section will override, or be overridden by, the list from the external file. The **Order=** keyword controls the order in which the lists are searched.

```
[LookUp Section]
Default       =      xxx
NoMatch       =      { I, D, W, E }
                     [ , { Default, Skip, Error } ]
Order         =      Query, File, List

LookUp Val    =      Val
LookUp Val 2  =      Val 2
   . . .

Query         =      @[ TagSvr ] Tag, Schema, Query$
QueryTarget   =      $Tag, Tgt-1$|$Tag, Fld-4$
QueryMulti    =      First | Error
```

If a list is coded in the section, the **Order=** keyword controls the list search sequence.

QueryTarget= is used to build a LookUp return string from the fields in the record and other references.

Query Specifies that an ARS Query will be used to select a record with which to build the returned string.

The **Query=** is coded exactly as in a Command Section (see *Query Statements* above)

If the **Query=** matches a record, the Tag is used to hold the loaded

values. These remain in memory until the next LookUp using the same LookUp section is processed.

QueryTarget Specifies how to build the string that will be returned when the Query matches a record.

Fields in the loaded record may be used to construct the returned string. That string can also use other references and the **$CTL, LookUp_Src$** reference.

QueryMulti This is an optional value than can be one of two keywords: **Error** or **First**. The default is

```
QueryMulti    =      Error
```

Normally, the Query specified should return exactly one record. If multiple records are returned this allows you to continue by loading the first record.

Note that Error will write an error message to the log but will not necessarily cause the LookUp to fail. The LookUp search string may still be found through other LookUp mechanisms. The **NoMatch=** keyword determines when to do when all coded LookUp mechanisms are exhausted.

Query records may be cached to avoid the overhead of issuing queries for the same record. The default is that the LookUp section does not use a cache. See Caching LookUp Records below for more about LookUp record caching.

Using an SQL Query

LookUp sections can also use direct SQL queries to build the translate string. The SQL statement is executed by the ARS server using the server's database credentials.

SQL Queries are similar to ARS Queries. The **QuerySql=** qualification string will generally have the source reference **$CTL, LookUp_Src$** in it.

A **QuerySql=** acts like a Load (but for an SQL row) that is allowed to fail. The row, if found, is made available to the rest of the script under the Query Tag given. The reference, **$CTL, LookUp$** may be used to determine if the **QuerySql=** was satisfied or a default or other list was used.

A string reference using fields of the LookUp record found is used to build the return string. That string reference is specified with the **QuerySqlTarget=** keyword.

If multiple rows match, you may throw an error (the default) or select the first record.

To use an SQL query add the **QuerySql=** keyword to the LookUp section.

When you use the **QuerySql=** keyword, all other related keywords are also required. Other LookUp mechanisms, including an internal list of value pairs, a list from an external file, an ARS Query will override, or be overridden by, the SQL query coded here. If a result is found, the SQL query is not executed at all.

The **Order=** keyword controls the order in which the LookUp mechanisms are searched.

```
[LookUp Section]
Default         =       xxx
NoMatch         =       { I, D, W, E }
                        [ , { Default, Skip, Error } ]
Order           =       QuerySql, List

LookUp Val      =       Val
LookUp Val 2    =       Val 2
   . . .

QuerySql        =       @[ TagSvr ] Tag, LookUpSqlFlds,   &
                        Select fld_a, fld_b from xxx      &
                        Where  fld_look_up =              &
                               '$CTL, LookUp_Src$'
QuerySqlTarget=         $Tag, fld_a$|$Tag, fld-b$
QuerySqlMulti=          First | Error

[LookUpSqlFlds]
fld_a           =       $       Subst /@/_/
fld_b           =       $       Date   julian
```

The **Order=** keyword controls the search sequence of LookUp mechanisms

The full features of **QuerySql=** can be used including field interpretation rules.

QuerySqlTarget= is used to build a LookUp return string from the columns in the row and other references.

QuerySql Specifies that an ARS server SQL Query will be used to select a row with which to build the returned string.

The **QuerySql=** is coded exactly as in a Command Section (see *Query SQL Statements* above)

If the **QuerySql=** matches a row, the Tag is used to hold the loaded values. These remain in memory until the next LookUp using the same LookUp section is processed.

QuerySqlTarget Specifies how to build the string that will be returned when the **QuerySql=** matches a row.

Columns in the loaded row may be used to construct the returned string.

That string can also use other references and the **$CTL, LookUp_Src$** reference. Column numbers, or, if specified, field names, can be used for the SQL row's columns..

QuerySqlMulti This is an optional value than can be one of two keywords: **Error** or **First**. The default is

```
QuerySqlMulti       =       Error
```

Normally, the **QuerySql=** specified should return exactly one row. If multiple rows are returned this allows you to continue by loading the first row.

Note that Error will write an error message to the log but will not necessarily cause the LookUp to fail. The LookUp search string may still be found through other LookUp mechanisms. The **NoMatch=** keyword determines when to do when all coded LookUp mechanisms are exhausted.

An SQL query may also be used in other ways not obvious in a LookUp function.

For example an SQL procedure may be coded in the SQL Query that manipulates a source string and returns a different string allowing you to write value transformation functions.

An SQL select count(*) may be used to assign to an integer field:

```
[Assignment Section]
Count         = @LookUp, LookUp Section, "dummy"

[LookUp Section]
QuerySql      =      Dmy-Tag, @na,                    &
                     Select count(*) from xxx         &
                     where  case_id =                 &
                            '$HpdSrc, Request ID$'
QuerySqlTarget=      $Dmy-Tag, 1$
```

This query makes no use of the LookUp source value instead using a different reference.

In this example an SQL statement is used to decrement a counter. Note that this SQL statement is Oracle specific.

```
[Assignment Section]
@Cmd          = Ref, MyVars, Ctr,                     &
                @LookUp, LookUp Section, "$MyVars, Ctr$"

[LookUp Section]
QuerySql      =      Dmy-Tag, @na,                    &
                     select                           &
                       to_number($CTL, LookUp_Src$-1) &
                     from dual
QuerySqlTarget=      $Dmy-Tag, 1$
```

QuerySql records may be cached to avoid the overhead of issuing queries for the same record. The default is that the LookUp section does not use a cache.

Caching LookUp Records

LookUp sections that issue **`Query=`** or **`QuerySql=`** queries can cache the records retrieved.

This only happens when the **`Cache=`** keyword is used in a LookUp section, and that section uses a **`Query=`** or a **`QuerySql=`**.

The **`Cache=`** keyword is used to specify a maximum cache size. Specifying 0 (zero) makes the cache unlimited.

When a LookUp section includes a cache, the source string is searched in the order given by that LookUp section against its internal list, file, or queries. Before the **`Query=`** or **`QuerySql=`** is executed, a search is made in the cache for the source string. If the string is found, the saved record is returned as though the Query or QuerySql had been executed.

This saves going to the ARS server for the query and record and can yield a significant performance boost.

Any Load= can be converted to use a LookUp to benefit from this performance boost.

It is important when using a `Cache=` to remember these things:

- Records are found in the cache according to the LookUp input string no matter what terms are used in the queries. Therefore, each LookUp source string that returns a record should always return that and only that record. No other source string should return that record.

  It is usually a simple matter to develop such a string. The string passed to the LookUp section does not have to be referenced by the LookUp queries. So, for example, the string could simply be the set of references that the LookUp uses as in this case:

```
@Cmd   =    @Ref,         MyVars,       LkupRslt,  &
            @LookUp,      LkUp-1,       "dmy"

@Cmd   =    @Ref,         MyVars,       LkupRslt,  &
            @LookUp,      LkUp-1,
            "$Tag1, field1$-$Tag2, field2$

[LkUp-1]
Cache  =    0
Query  =    Tag,                                                  &
            Schema,                                               &
            'Field 1'   = $Tag1, field1$    and                   ^
            'Field 2'   = $Tag2, field2$
```

Cannot use a Cached LookUp

Can use a Cached LookUp

- If a record that is returned by a LookUp will be updated through the script (or through any other means) then a cache should not be used. This will ensure that when needed the record will contain the updated values.

- If two different LookUp sessions can return the same record ***and*** use the same Tag for that record, then a cache cannot be used in either LookUp.

Using different LookUp Lists

To use different LookUp sections to make a match, simply use several different LookUp sections in a sequence of assignment so that the assignments are run only if the LookUp previously has failed.

The condition can be on the $CTL, LookUp$ reference or on the assigned value if a default was selected.

```
@Cmd   =    @Ref,         MyVars,       LkupRslt,                  &
              @LookUp,    LkUp-1,       $MyVars, LkupSrc$
@Cmd   =    @if ("$CTL, LookUp$" == "Defaul$")                       &
            @Ref,         MyVars,       LkupRslt,                  &
              @LookUp,    LkUp-2,       $MyVars, LkupSrc$
@Cmd   =    @if ("$CTL, LookUp$" == "Defaul$")                       &
            @Ref,         MyVars,       LkupRslt,                  &
              @LookUp,    LkUp-3,       $MyVars, LkupSrc$
```

In this example, the LookUp section returns the original source string by default and the three different sections are always run. In fact, if the Default for the LookUp is the original string, the example above and below are equivalent.

```
@Cmd   =    @Ref,         MyVars,       LkupRslt,                  &
              @LookUp,    LkUp-1,       $MyVars, LkupSrc$
@Cmd   =    @Ref,         MyVars,       LkupRslt,                  &
              @LookUp,    LkUp-2,       $MyVars, LkupRslt$
@Cmd   =    @Ref,         MyVars,       LkupRslt,                  &
              @LookUp,    LkUp-3,       $MyVars, LkupRslt$
```

Field Type Notes

Diary Fields

Diary Field values can be one of two types: a character string or a formatted diary field string.

A formatted diary field string contains diary field entries including a time stamp, a user, and the text of the entry. These are referenced by using a loaded record's diary field data.

When a diary field contains a formatted diary string, no concatenation is permitted. That string can only be used in a new create, not an update. The actual update is made using the Merge API after the record is created normally.

Normal strings can be assigned to diary entries on both new creates and updates. They can also be concatenated in the Assignments file. In these cases, the string, the current time, and the user that Meta-Update signs on with, are appended as a new entry to the current diary field contents.

Currency Fields

Currency Fields can be specified as a specially constructed string. This is the same whether the currency field value is in an import file or used in a literal assignment. The field definition on the form plays a role in the assignment of currency fields.

To specify a currency field, the minimum required is a decimal numeric quantity. If desired, all functional currencies may be specified as well as a date for the conversion of the functional currencies.

The syntax for a currency field value is:

nnn.nn [XXX] [date] [nnn.nn XXX] …

where	nnn.nn	is a sequence of digits with an optional decimal point decimal portion
	XXX	is an ISO currency code allowed in the field being assigned
	date	is a formatted date value yyyy [.mm [dd [hh [:mm [:ss]]]]] See date fields above for more information.
	nnn.nn	the value part for a functional currency
	XXX	the ISO currency code for a functional currency

Examples:

123.62
123.62 EUR
123.62 EUR 2005.10.01 14:30 136.89 USD 174.29 CAD
123.62 EUR 2005.10.01 136.89 USD 174.29 CAD

It is an error to specify a currency code that is not defined as being permitted for the field. As of release 6.3 or ARS, this is an error not caught and results in a null assignment to the field. Meta-Update catches this error and does not attempt the assignment or the update. An Error message is produced and the update fails.

Numeric Fields

Numeric Fields are specified as an optional leading sign indicator and a sequence of digits. Integers can only have digits. Real numbers may have a decimal point and more digits.

Numeric constants in assignment sections must be specified in the expected format. References from ARS records are always in the correct format. References from CSV files must have their values transposed into the expected formats.

For data is in a CSV file, the Subst formatting option can be used to remove thousands separators and to convert the decimal point into a period when required. For example to convert a decimal value in German notation to the internal Meta-Update representation:

```
Numerc Value        =      $      Subst /.//    Subst /,///
```

A value of “1.234,56” would then be transposed into “1234.56”

Enum or Selection Fields

Selection Fields are stored in the database as integers. There are three different types of selection fields defined in the API since release 5 though the administrator tool prior to release 7 only allowed a single type. Now, with the advent of release 7, the admin tool supports two types though a third type is supported with the API.

The two types are sequentially enumerated types and enumerated types with gaps.

Meta-Update takes character strings or integers for enumerated values. References are converted to character strings. So a “Closed” value from one schema will match a “Closed” value in another schema even if the underlying numerical values are different. A value of “10” will first be searched though the aliases and if not found will be accepted as an integer value.

Values must be defined in the field or an error is thrown.

Date Fields

Meta-Update holds date values internally as character strings of the form ±yyyymmddhhmmss in the local time zone. It converts to and from ARS date data types as needed.

ARS Date fields hold a date from Jan 1, 4912 BC through Jan 1, 9999. The value is represented by an integer number of days since the 4713 BC date. The time component of a date field is ignored. There are no time zone adjustments.

Date/Time Fields

ARS date / time fields hold a date and time stamp in a number of seconds from Jan 1, 1970. They are always saved in the GMT or UT time zone.

Meta-Update converts ARS Date / Time fields into a character string representing the local time on the machine that Meta-Update is running on when-ever any record with such fields is read. Similarly, these character strings are converted back when inserting into an ARS field for updating.

Meta-Update date strings are as follows:

yyyy/mm/dd hh:mm:ss	
yyyy-mm-dd hh:mm:ss	
yyyy.mm.dd hh:mm:ss	
yyyymmddhhssmm	
$date$	represents current date / time
$time$	represents current date / time
$daystart$	represents current date at 00:00:00
$dayend$	represents current date at 23:59:59

Any missing components will be treated as if they were zero (one for month and day).

File records can specify date columns' formats if different than above. Meta-Update then converts from the file's strings into the above. Any file date field, can be assigned to any ARS date field.

Diary loops supply the entry date in various different date formats as different references. For assignments, you'll need to use the normal reference, $DiaryTag, Date$ which represents the date as above.

In assignments, date fields can be references to ARS fields, file fields, or strings. No reformatting of dates is required as all internal dates have the same format as described above. When assigning constants, the constants need to conform to the above format.

When Queries are coded, ARS expects a date to be formatted according to the current machine's international configuration. Queries using date fields with compare values from diary loops can use the specific reference for that machine's settings.

The ARDATE environment variable may be set before Meta-Update is started to alter how the ARS API interprets dates in queries. Any change to the ARDATE environment variable within a Meta-Update script have no effect on the ARS API, so if you decide to use this, it must be set before the Meta-Update job is fired.

Full documentation on the ARDATE specification is given by the BMC Remedy documents. The release 7.6.03 documents specifically say that ARDATE has no effect on clients. However, testing with Meta-Update scripts has proven that if ARDATE is set before Meta-Update begins, then dates in Queries are interpreted according to the ARDATE setting.

If the same Meta-Update script is to be run on different machines with different regional settings and dates are specified in Queries, then it is a good idea to set the ARDATE environment variable so that the queries will be interpreted in the same way no matter the regional settings on the machine running Meta-Update.

The format of the ARDATE value differs for UNIX and Windows.

These summaries of ARDATE syntax are taken from the BMC Action Request System 7.6.03 Form and Application Objects document.

This table lists the UNIX field descriptors that you can use with ARDATE, ARDATEONLY, and ARTIMEONLY.

Descriptor	Function
%%	Same as %
%a	Day of week using locale's abbreviated weekday names
%A	Day of week using locale's full weekday names
%b	or %h Month using locale's abbreviated month names
%B	Month using locale's full month names
%d	Day of month (01–31)
%D	Date as %m/%d/%y
%e	Day of month (1–31; single digits are preceded by a blank)
%H	Hour (00–23)
%I	Hour (00–12)
%k	Hour (0–23; single digits preceded by a blank)—Sun Solaris™ operating system only
%m	Month number (01–12)
%M	Minute (00–59)
%p	Locale's equivalent of a.m. or p.m., whichever is appropriate
%r	Time as %I:%M:%S %p
%R	Time as %H:%M
%S	Seconds (00–59)
%T	Time as %H:%M:%S
%w	Day of week (Sunday is day 0)
%x	Date, using locale's date format
%X	Time, using locale's time format
%y	Year within century (00–99)
%Y	Year, including century (for example, 2004)

Table 1 ARDATE Field Descriptors for UNIX

This table lists the Windows field descriptors that you can use with ARDATE, ARDATEONLY, and ARTIMEONLY.

Time notations	Displays
h	Hour (hh displays the hour with a leading zero)
m	Minute (mm displays the minute with a leading zero)
s	Second (ss displays the second with a leading zero)
tt	A.M. or P.M.
h/H	12 or 24 hour time display
Date notations	**Displays**
d, dd	Day
ddd, dddd	Day of the week
M	Month

y	Year

Table 2 ARDATE Field Descriptors for Windows

When the value is from a reference to an ARS field, the date needs to be rearranged for the query in the appropriate manor.

The following sample code, will take a date field reference, and create a new field to hold an ARS query date value in the German format:

```
@Cmd            = Ref, V, asg-Date-split, @regex,                    &
        '([0-9]*)/([0-9]*)/([0-9]*) ([0-9]*):([0-9]*):([0-9]*)',
        $RecXx, Create Date$
@Cmd            = Ref, V, Date-Fff,                                  &
        "$V, dy$/$V, mn$/$V, yr$ $V, hr$:$V, mm$:$V, ss$"
[asg-DoPpl-yr]
yr              = $
mn              = $
dy              = $
hr              = $
mm              = $
ss              = $
```

You can include the above assignments in an AssignInit processed before a `Query=` and the reference the date as $V, Date-Fff$ instead of $RecXxx, Submitter$.

Attachment Fields

In Remedy, there are two attributes for an attachment value: the name of the attachment, and the file name of the attachment.

With the Remedy GUI (through the User tool or a browser), assigning an attachment sets both the name of the attachment and the file name equal. Saving an attachment allows you to create a file of any name.

With Meta-Update, it is possible that the real file name that is the source of a file to be attached is not equal to the file name desired in the ARS data. Consider the case where Meta-Update is running on a server and the file needs be opened on the client.

In Meta-Update, attachment field values can be specified as a string, as two strings, or as a reference to another attachment field. Or, the three "forms" of an attachment assignment are:

```
1   Att-Fld         =     "attachment file name, os file name"
2   Att-Fld         =     attachment (& os) file name
3   Att-Fld         =     Ta,  field
```

This example is introduced and discussed below. The three different "forms" of an attachment value are specified.

```
1   Att-Fld         =     "C:\tmp\logos.jpg, C:\temp\attach-1.dta"
2   Att-Fld         =     C:\tmp\logos.jpg
3   Att-Fld         =     Src,  AttFld
```

For 1), If the file `C:\temp\attach-1.dta` exists and is readable, and,
for 2) if the file `C:\tmp\logos.jpg` exists and is readable, and,
for 3) if the reference `$Src, AttFld$` is a loaded ARS record with an attachment field containing `C:\tmp\logos.jpg`

then, the three statements above are almost equivalent and result in an attachment named C:\tmp\logos.jpg being assigned to the target attachment field.

For the three statements, these files need to exist:

1 `C:\temp\attach-1.dta`
2 `C:\tmp\logos.jpg`
3 none

When an attachment field is assigned the contents of another ARS attachment value through a reference, as in example 3 above, the attachment value itself is retrieved, resulting in an additional API call to the ARS server.

When Meta-Update reads an ARS record through the ARS API, (as in the read that populated the tag, **`Src`**), the descriptive contents of attachment fields are read. The attachment itself is not read until required.

When an attachment field is assigned the value of another attachment field through a reference, and the source value is non-null, Meta-Update retrieves the contents of the attachment. That retrieval is made using a memory buffer. The buffer is freed when the original record is freed, in the above example's case, when the **`Src`** tag is reloaded.

When an attachment field is a string value, that string value (in either one or two components) refers to a file name.

That file name must be able to be read by the Remedy API. That is, the file must exist, and the user running the Meta-Update binary should have read rights to the file. Generally speaking, you should be able to open that file with the Windows Explorer if you are running Windows.

Predefined Reference Tags

Meta-Update automatically defines some reference tags. These tags may be used anywhere that any other tags can be used. These tags are available to the script on start-up:

CTL Meta-Update and script information
Arg Script defined arguments
ENV Environment variables
AR_INFO AR Server Information
RdSvr_AR_INFO AR Server Information for the Read Server with the Tag "RdSvr"
As many of these are defined as there are Read Servers.
CTL-*RdSvr_Schema* When any schema is loaded, queried, or updated, an automatic tag is created that holds information about the schema.

The **ENV**, **AR_INFO**, and **RdSvr_AR_INFO** tags may be assigned values with the **@Cmd, Reference** assignment command.

CTL – Meta-Update Information

The CTL tag gives Meta-Update wide operational information:

Tag	Field	Value
CTL	Server	The Server where the update will occur.
CTL	Port	The Port number for the update server connection (or zero).
CTL	RPC	The RPC number for the update server connection (or zero).
CTL	User	The User ID being used for Meta-Update updates.
CTL	Password	The Password being used for Meta-Update updates.
CTL	Script	The script file and path being run. E.g. ***./path/x.ini***
CTL	ScriptF	The script file being run: minus path information. E.g. ***x.ini***
CTL	ScriptFx	The script file being run: minus path information and extension. E.g.: ***x***
CTL	OS	One of: UNIX, LINUX, or, Windows.
CTL	ArsVer	The ARS Server version whole number. E.g. 7 for ARS 7.01.
CTL	ArsVerS	The ARS Server "Sub-version". E.g. 1 for ARS 7.01.x.
CTL	Pid	The decimal process ID of the Meta-Update run.
CTL	Pidx	The hexadecimal (lower case) process ID.
CTL	PidX	The hexadecimal (upper case) process ID.
CTL	Schema	The Schema being updated.
CTL	SchemaFnm	The CTL, Schema transposed into a simple file name containing no special characters. Note that it is not possible for two similar schema names to be transposed into the same file name. Each special character is translated into an underscore. This is generally also the database view name automatically created by ARS 6 and above. Exceptions would be some specific tables that are SQL keywords, like USER, and tables beginning with numbers.
CTL	Operation	One of "Update", "Create".
CTL	ID	The ID for the record being updated or "" when creating.

The following tags are section based. That is a different tag exists for each launched section. The tag name is the prefix "CTL-" and the section name. For example, the section [DoFileImp] will automatically define the reference: CTL-DoFileImp. This reference will be available in that section and any sections launched from that section. When that section completes the symbol will no longer be defined.

CTL-***Sec***	Rec	The current record being processed within a query or file. Null (the empty string) if not processing a File= or Query.

Some tags are only defined when appropriate. For example, the Record counters count records in either a query or a file. A file cannot reasonably have a maximum defined so that counter is unavailable

Arg – Program Arguments

The **Arg** tag holds any program arguments as defined by the **ArgNm=** keyword in the **Main** section.

All arguments defined with the **ArgNm=** keyword are defined when the Meta-Update script starts, whether or not the command line included a value for an argument. If the command line did not include a value for a given argument, it will contain an empty string equivalent to $NULL$.

ENV – The Environment

The **ENV** Tag refers to the environment. The fields in the environment are initialised when Meta-Update begins and reflect the environment of the shell used to start Meta-Update.

Fields of the environment are case sensitive. **$ENV, Path$** does not yield the standard PATH variable. **ENV, PATH** does.

Assignments to the environment may be made through the Reference assignment command.

When a Reference command sets a variable in the ENV tag, the Meta-Update environment is adjusted.

Any changes in the environment will be reflected in subsequent references within Meta-Update and within any client processes that Meta-Update starts. This includes any client processes launched by the Meta-Update Spawn command.

When Meta-Update completes, the environment will revert to what it was when the Meta-Update process was started.

AR_INFO – ARS Server Information

The **AR_INFO** Tag refers to the ARS Server Information values available.

The fields in this tag are initialized when Meta-Update begins and reflect the values returned by the main Update Server. The number of fields available is the minimum of the API version being used by Meta-Update and the ARS server version.

Field names are the AR_SERVER_INFO_xxx defines in the API file, ar.h

The number of fields defined vary by the release of the server.

*RdSvr*_AR_INFO – ARS Server Information

The ***RdSvr*_AR_INFO** Tag refers to the ARS Server Information values available for a read server with the tag, "**RdSvr**".

AR_INFO – Table of Fields and Values

The following table lists the AR_INFO tag's field names, the ARS Server version that introduced the field, typical values, and whether the field is readable or writable.

The values were taken from an OOTB 7.6.04 patch 2 server installed on a Window 2003 Server X64 Standard VM. In some cases, the value has been truncated.

Release Introduced	Name	RW	Example Value
5.12	DB_TYPE	R	SQL -- SQL Server
	SERVER_LICENSE	R	Server
	FIXED_LICENSE	R	18
	VERSION	R	7.6.04 Build 002 201101141059
	ALLOW_GUESTS	RW	1
	USE_ETC_PASSWD	RW	0
	XREF_PASSWORDS	RW	0
	DEBUG_MODE	RW	1179711
	DB_NAME	R	ARSystem
	DB_PASSWORD	W	
	HARDWARE	R	x86_64
	OS	R	Windows Server 2003
	SERVER_DIR	R	D:\Apps\BMC\ARSystem\ARServer\Db\
	DBHOME_DIR	R	
	SET_PROC_TIME	RW	5
	EMAIL_FROM	RW	ARSystem
	SQL_LOG_FILE	RW	E:\Logs-ARS\a001.log
	FLOAT_LICENSE	R	0
	FLOAT_TIMEOUT	RW	2
	UNQUAL_QUERIES	RW	1
	FILTER_LOG_FILE	RW	E:\Logs-ARS\a001.log
	USER_LOG_FILE	RW	E:\Logs-ARS\a001.log
	REM_SERV_ID	R	
	MULTI_SERVER	RW	1
	EMBEDDED_SQL	R	0
	MAX_SCHEMAS	R	0
	DB_VERSION	R	2008 R2 (SP1) - 10.50.2500.0 (X64)
	MAX_ENTRIES	RW	0
	MAX_F_DAEMONS	RW	12
	MAX_L_DAEMONS	RW	8
	ESCALATION_LOG_FILE	RW	E:\Logs-ARS\a001.log
	ESCL_DAEMON	RW	1
	SUBMITTER_MODE	RW	2
	API_LOG_FILE	RW	E:\Logs-ARS\a001.log
	FTEXT_FIXED	R	1
	FTEXT_FLOAT	R	1
	FTEXT_TIMEOUT	RW	2
	RESERV1_A	RW	0
	RESERV1_B	RW	0
	RESERV1_C	RW	0
	SERVER_IDENT	R	0050560C63F6

	DS_SVR_LICENSE	RW	Server
	DS_MAPPING	R	Distributed Mapping
	DS_PENDING	R	Distributed Pending
	DS_RPC_SOCKET	RW	
	DS_LOG_FILE	RW	E:\Logs-ARS\a001.log
	SUPPRESS_WARN	RW	
	HOSTNAME	R	sthvmwin2003
	FULL_HOSTNAME	R	sthvmwin2003
	SAVE_LOGIN	RW	
	U_CACHE_CHANGE	R	1332947367
	G_CACHE_CHANGE	R	1332944611
	STRUCT_CHANGE	R	1348325258
	CASE_SENSITIVE	RW	1
	SERVER_LANG	R	ENU;UTF-8
	ADMIN_ONLY	RW	0
	CACHE_LOG_FILE	RW	
	FLASH_DAEMON	RW	0
	THREAD_LOG_FILE	RW	E:\Logs-ARS\a001.log
	ADMIN_TCP_PORT	RW	
	ESCL_TCP_PORT	RW	0
	FAST_TCP_PORT	RW	0
	LIST_TCP_PORT	RW	0
	FLASH_TCP_PORT	RW	0
	TCD_TCP_PORT	RW	0
	DSO_DEST_PORT	RW	
	INFORMIX_DBN	R	
	INFORMIX_TBC	R	
	INGRES_VNODE	RW	
	ORACLE_SID	R	
	ORACLE_TWO_T	R	
	SYBASE_CHARSET	R	
	SYBASE_SERV	R	STHVMWIN2003
	SHARED_MEM	RW	
	SHARED_CACHE	RW	
	CACHE_SEG_SIZE	RW	
	DB_USER	R	ARAdmin
	NFY_TCP_PORT	RW	
	FILT_MAX_TOTAL	RW	500000
	FILT_MAX_STACK	RW	10000
	DEFAULT_ORDER_BY	RW	1
	DELAYED_CACHE	RW	0
	DSO_MERGE_STYLE	RW	0
	EMAIL_LINE_LEN	RW	1024
	EMAIL_SYSTEM	RW	
	INFORMIX_RELAY_MOD	R	
	PS_RPC_SOCKET	RW	390601:1 1 ;390603:1 1 ;390620:2 12 ;390621:5 16 ;390635:2 8 ;390680:2 2 ;
	REGISTER_PORTMAPPER	RW	1
	SERVER_NAME	RW	sthvmwin2003
	DBCONF	R	
	APPL_PENDING	R	Application Pending

	AP_RPC_SOCKET	RW	390680
	AP_LOG_FILE	RW	
	AP_DEFN_CHECK	RW	
	MAX_LOG_FILE_SIZE	RW	0
	CLUSTERED_INDEX	RW	1
	ACTLINK_DIR	RW	
	ACTLINK_SHELL	RW	
	USER_CACHE_UTILS	RW	1
	EMAIL_TIMEOUT	RW	10
	EXPORT_VERSION	R	11
	ENCRYPT_AL_SQL	RW	0
	SCC_ENABLED	RW	
	SCC_PROVIDER_NAME	RW	
	SCC_TARGET_DIR	RW	
	SCC_COMMENT_CHECKIN	RW	
	SCC_COMMENT_CHECKOUT	RW	
	SCC_INTEGRATION_MODE	RW	
	EA_RPC_SOCKET	RW	
	EA_RPC_TIMEOUT	RW	
	USER_INFO_LISTS	RW	128
	USER_INST_TIMEOUT	RW	7200
	DEBUG_GROUPID	RW	1
	APPLICATION_AUDIT	RW	
	EA_SYNC_TIMEOUT	RW	300
	SERVER_TIME	RW	1348555192
	SVR_SEC_CACHE	RW	0
	LOGFILE_APPEND	RW	0
	MINIMUM_API_VER	RW	0
	MAX_AUDIT_LOG_FILE_SIZE	RW	0
	CANCEL_QUERY	RW	1
	MULT_ASSIGN_GROUPS	RW	1
	ARFORK_LOG_FILE	RW	D:\Apps\BMC\ARSystem\ARServer\Db\arfork.log
	DSO_PLACEHOLDER_MODE	RW	0
	DSO_POLLING_INTERVAL	RW	
	DSO_SOURCE_SERVER	RW	
	DS_POOL	RW	Distributed Pool
	DSO_TIMEOUT_NORMAL	RW	
	ENC_PUB_KEY		
	ENC_PUB_KEY_EXP	RW	86400
	ENC_DATA_KEY_EXP	RW	2700
	ENC_DATA_ENCR_ALG	RW	1
	ENC_SEC_POLICY	RW	2
	ENC_SESS_H_ENTRIES	RW	509
	DSO_TARGET_CONNECTION	RW	
	PREFERENCE_PRIORITY	RW	0
	ORACLE_QUERY_ON_CLOB	RW	
	MESSAGE_CAT_SCHEMA	R	AR System Message Catalog
	ALERT_SCHEMA	RW	Alert Events
	LOCALIZED_SERVER	RW	1
	SVR_EVENT_LIST	RW	1;

	DISABLE_ADMIN_OPERATIONS	RW	0
	DISABLE_ESCALATIONS	RW	0
	ALERT_LOG_FILE	RW	E:\Logs-ARS\a001.log
	DISABLE_ALERTS	RW	0
	CHECK_ALERT_USERS	RW	0
	ALERT_SEND_TIMEOUT	RW	7
	ALERT_OUTBOUND_PORT	RW	0
	ALERT_SOURCE_AR	RW	AR
	ALERT_SOURCE_FB	RW	FB
	DSO_USER_PASSWD	RW	
	DSO_TARGET_PASSWD	RW	
	APP_SERVICE_PASSWD	RW	
	MID_TIER_PASSWD	RW	
	PLUGIN_LOG_FILE	RW	E:\Logs-ARS\a001.log
	SVR_STATS_REC_MODE	RW	0
	SVR_STATS_REC_INTERVAL	RW	60
	DEFAULT_WEB_PATH	RW	http://sthvmwin2003:8080/arsys
	FILTER_API_RPC_TIMEOUT	RW	180
	DISABLED_CLIENT	RW	
	PLUGIN_PASSWD	RW	
	PLUGIN_ALIAS	RW	ARSYS.ARF.REGISTRY ARSYS.ARF.REGISTRY sthvmwin2003:9999;ARSYS.ARDBC.REGISTRY ARSYS.ARDBC.REGISTRY sthvmwin2003:9999;ARSYS.ARDBC.ARREPORTENGINE ARSYS. ARDBC.ARREPORTE
	PLUGIN_TARGET_PASSWD	RW	
	REM_WKFLW_PASSWD	RW	
	REM_WKFLW_TARGET_PASSWD	RW	
	EXPORT_SVR_OPS	RW	
	INIT_FORM	RW	
	ENC_PUB_KEY_ALG	RW	4
	IP_NAMES	RW	
	DSO_CACHE_CHK_INTERVAL	RW	
	DSO_MARK_PENDING_RETRY	RW	0
	DSO_RPCPROG_NUM	RW	
	DELAY_RECACHE_TIME	RW	5
	DFLT_ALLOW_CURRENCIES	RW	
	CURRENCY_INTERVAL	RW	60
	ORACLE_CURSOR_SHARE	RW	
	DB2_DB_ALIAS	R	
	DB2_SERVER	R	
	DFLT_FUNC_CURRENCIES	RW	
	EMAIL_IMPORT_FORM	RW	0
	EMAIL_AIX_USE_OLD_EMAIL	RW	0
	TWO_DIGIT_YEAR_CUTOFF	RW	2041
	ALLOW_BACKQUOTE_IN_PROCESS	RW	0
	DB_CONNECTION_RETRIES	RW	101
6	DB_CHAR_SET	R	utf-16
	CURR_PART_VALUE_STR	R	VALUE
	CURR_PART_TYPE_STR	R	TYPE
	CURR_PART_DATE_STR	R	DATE

	HOMEPAGE_FORM	RW	AR System Customizable Home Page
	DISABLE_FTS_INDEXER	RW	0
	DISABLE_ARCHIVE	RW	0
	SERVERGROUP_MEMBER	RW	0
	SERVERGROUP_LOG_FILE	RW	E:\Logs-ARS\a001.log
	FLUSH_LOG_LINES	RW	1
	SERVERGROUP_INTERVAL	RW	60
	JAVA_VM_OPTIONS	RW	
	PER_THREAD_LOGS	RW	0
	CONFIG_FILE	R	D:\Apps\BMC\ARSystem\conf\ar.cfg
	SSTABLE_CHUNK_SIZE	RW	1000
	SG_EMAIL_STATE	R	0
	SG_FLASHBOARDS_STATE	R	0
	SERVERGROUP_NAME	RW	
	SG_ADMIN_SERVER_NAME	RW	
	LOCKED_WKFLW_LOG_MODE	RW	0
	ROLE_CHANGE	RW	1295305780
	SG_ADMIN_SERVER_PORT	RW	
6.3	PLUGIN_LOOPBACK_RPC	RW	390626
	CACHE_MODE	RW	0
	DB_FREESPACE	RW	6171296
	GENERAL_AUTH_ERR	RW	1
	AUTH_CHAINING_MODE	RW	0
	RPC_NON_BLOCKING_IO	RW	0
	SYS_LOGGING_OPTIONS	RW	0
	EXT_AUTH_CAPABILITIES	RW	0
	DSO_ERROR_RETRY	RW	
7.0	PREF_SERVER_OPTION	RW	1
	FTINDEXER_LOG_FILE	RW	E:\Logs-ARS\a001.log
	EXCEPTION_OPTION	RW	0
	ERROR_EXCEPTION_LIST	RW	
	DSO_MAX_QUERY_SIZE	RW	
	ADMIN_OP_TRACKING	RW	0
	ADMIN_OP_PROGRESS	R	
	PLUGIN_DEFAULT_TIMEOUT	RW	600
	EA_IGNORE_EXCESS_GROUPS	RW	1
	EA_GROUP_MAPPING	RW	
	PLUGIN_LOG_LEVEL	RW	100
	FT_THRESHOLD_LOW	RW	200
	FT_THRESHOLD_HIGH	RW	1000000
	NOTIFY_WEB_PATH	RW	
	DISABLE_NON_UNICODE_CLIENTS	RW	0
	FT_COLLECTION_DIR	RW	D:\Apps\BMC\ARSystem\ftsconfiguration\collection
	FT_CONFIGURATION_DIR	RW	D:\Apps\BMC\ARSystem\ftsconfiguration\conf
	FT_TEMP_DIR	RW	
	FT_REINDEX	RW	0
	FT_DISABLE_SEARCH	RW	0
	FT_CASE_SENSITIVITY	R	1
	FT_SEARCH_MATCH_OP	RW	4

7.5	FT_STOP_WORDS	RW	a;about;above;across;after;again;against;all;almost;alone;along;already;also;although;always;among;an;and;another;any;anybody;anyone;anything;anywhere;are;area;areas;around;as;ask;asked;at;away;b;back;be;became;because;become;been;before;began;behind;bei 
	FT_RECOVERY_INTERVAL	RW	60
	FT_OPTIMIZE_THRESHOLD	RW	1000
	MAX_PASSWORD_ATTEMPTS	RW	0
	GUESTS_RESTRICT_READ	RW	0
	ORACLE_CLOB_STORE_INROW	RW	
	NEXT_ID_BLOCK_SIZE	RW	100
	NEXT_ID_COMMIT	RW	0
	RPC_CLIENT_XDR_LIMIT	RW	0
	CACHE_DISP_PROP	RW	3
	USE_CON_NAME_IN_STATS	RW	0
	DB_MAX_ATTACH_SIZE	RW	0
	DB_MAX_TEXT_SIZE	RW	2147483647
	GUID_PREFIX	RW	
	MULTIPLE_ARSYSTEM_SERVERS	RW	1
	ORACLE_BULK_FETCH_COUNT	RW	50
	MINIMUM_CMDB_API_VER	RW	3
	PLUGIN_PORT	RW	
	PLUGIN_LIST	RW	ardbcconf.dll; reportplugin.dll; ServerAdmin.dll; FlashboardObject.dll; "D:\Apps\BMC\ARSystem\arealdap\arealdap.dll"; "D:\Apps\BMC\ARSystem\ardbcldap\ardbcldap.dll"; "D:\Apps\BMC\ARSystem\approval\bin\arapprove.dll"; "D:\Apps\BMC\BMC Service Level Management\bin\omfbjiefilapidll"; "D:\Apps\BMC\BMC ServiceLevelManagement\bin\arfslasetup.dll" 
	PLUGIN_PATH_LIST	RW	D:\Apps\BMC\ARSystem; D:\Apps\BMC\ARSystem\pluginsvr; D:\Apps\BMC\ARSystem\arealdap; D:\Apps\BMC\ARSystem\ardbcldap; D:\Apps\BMC\AtriumCore\cmdb\server64\bin; D:\Apps\BMC\BMC Service Level Management\bin
	SHARED_LIB	RW	cmdbsvr7604_win64.dll
	SHARED_LIB_PATH	RW	D:\Apps\BMC\AtriumCore\cmdb\server64\bin
	CMDB_INSTALL_DIR	RW	D:\Apps\BMC\AtriumCore\cmdb
	RE_LOG_DIR	RW	D:\Apps\BMC\AtriumCore\Logs
	LOG_TO_FORM	RW	0
	SQL_LOG_FORM	RW	AR System Log: SQL
	API_LOG_FORM	RW	AR System Log: API
	ESCL_LOG_FORM	RW	AR System Log: Escalation
	FILTER_LOG_FORM	RW	AR System Log: Filter
	USER_LOG_FORM	RW	AR System Log: User
	ALERT_LOG_FORM	RW	AR System Log: Alert
	SVRGRP_LOG_FORM	RW	AR System Log: Server Group
	FTINDX_LOG_FORM	RW	AR System Log: FullText Index
	THREAD_LOG_FORM	RW	AR System Log: Thread

	FIPS_SERVER_MODE	RW	Disabled
	FIPS_CLIENT_MODE	RW	Disabled
	FIPS_STATUS	RW	Disabled
	ENC_LEVEL	RW	Standard
	ENC_ALGORITHM	RW	Disabled
	FIPS_MODE_INDEX	RW	0
	FIPS_DUAL_MODE_INDEX	RW	0
	ENC_LEVEL_INDEX	RW	-1
	DSO_MAIN_POLL_INTERVAL	RW	
	RECORD_OBJECT_RELS	RW	0
	LICENSE_USAGE	RW	0
	COMMON_LOG_FORM	RW	AR System Log: ALL
	LOG_FORM_SELECTED	RW	0
	MAX_CLIENT_MANAGED_TRANSACTIONS	RW	0
	CLIENT_MANAGED_TRANSACTION_TIMEOUT	RW	60
	OBJ_RESERVATION_MODE	RW	0
	NEW_ENC_PUB_KEY_EXP	RW	86400
	NEW_ENC_DATA_KEY_EXP	RW	2700
	NEW_ENC_DATA_ALG	RW	0
	NEW_ENC_SEC_POLICY	RW	2
	NEW_FIPS_SERVER_MODE	RW	Invalid Option
	NEW_ENC_LEVEL	RW	Disabled
	NEW_ENC_ALGORITHM	RW	Disabled
	NEW_FIPS_MODE_INDEX	RW	0
	NEW_ENC_LEVEL_INDEX	RW	-1
	NEW_ENC_PUB_KEY	RW	Disabled
	CUR_ENC_PUB_KEY	RW	Disabled
	NEW_ENC_PUB_KEY_INDEX	RW	0
	CURRENT_ENC_SEC_POLICY	RW	Disabled
	ENC_LIBRARY_LEVEL	RW	1
	NEW_FIPS_ALG	RW	0
	FIPS_ALG	RW	AES-128
	FIPS_PUB_KEY	RW	RSA-1024
	WFD_QUEUES	RW	
	VERCNTL_OBJ_MOD_LOG_MODE	RW	0
	MAX_RECURSION_LEVEL	RW	25
	FT_SERVER_NAME	RW	
	FT_SERVER_PORT	RW	
	VERCNTL_OBJ_MOD_LOG_SAVE_DEF	RW	0
	SG_AIE_STATE	R	0
	MAX_VENDOR_TEMP_TABLES	RW	1
	DSO_LOG_LEVEL	RW	0
	DS_PENDING_ERR	RW	Distributed Pending Errors
	REGISTRY_LOCATION	RW	
	REGISTRY_USER	RW	
	REGISTRY_PASSWORD	RW	
	DSO_LOG_ERR_FORM	RW	0
	ARSIGNALD_LOG_FILE	RW	E:\Logs-ARS\a001.log
	FIRE_ESCALATIONS	RW	

	PRELOAD_NUM_THREADS	RW	20
	PRELOAD_NUM_SCHEMA_SEGS	RW	300
	PRELOAD_THREAD_INIT_ONLY	RW	1
7.6	CREATE_WKFLW_PLACEHOLDER	RW	0
	MFS_TITLE_FIELD_WEIGHT	RW	1
	MFS_ENVIRONMENT_FIELD_WEIGHT	RW	1
	MFS_KEYWORDS_FIELD_WEIGHT	RW	1
	COPY_CACHE_LOGGING	RW	0
	DSO_SUPPRESS_NO_SUCH_ENTRY_FOR_DELETE	RW	0
	USE_FTS_IN_WORKFLOW	RW	1
	MAX_ATTACH_SIZE	RW	0
	DISABLE_ARSIGNALS	RW	0
	FT_SEARCH_THRESHOLD	RW	10000
	REQ_FIELD_IDENTIFIER	RW	*
	REQ_FIELD_IDENTIFIER_LOCATION	RW	1
	FT_SIGNAL_DELAY	RW	10
	ATRIUM_SSO_AUTHENTICATION	RW	0
	OVERLAY_MODE	RW	1
	FT_FORM_REINDEX	RW	
7.6.04	DS_LOGICAL_MAPPING	RW	Distributed Logical Mapping
	DB_CONNECTION_TIMEOUT	RW	30
	ATRIUMSSO_LOCATION	RW	
	ATRIUMSSO_USER	RW	
	ATRIUMSSO_PASSWORD	RW	
	SUPPRESS_DOMAIN_IN_URL	RW	0
	RESTART_PLUGIN	W	
	USE_PROMPT_BAR_FOR	W	
	ATRIUMSSO_KEYSTORE_PATH	RW	
	ATRIUMSSO_KEYSTORE_PASSWORD	RW	

CTL – Schema Tag

Any Remedy form queried, loaded, or updated by a script causes a new tag to be created.

This Tag holds information about the form.

The Tag created includes the read server tag if the load or query was from a read server and the schema name itself (including spaces and special characters).

Field	Value	Description	""
TypeSchema	Regular, Join, View, Dialog, Vendor	ARS Schema Type	""
Join	bool	TypeSchema is Join	0
Join1	string	Join schema 1	""
Join2	string	Join schema 2	""

View	bool	TypeSchema is View	0
ViewName	string	the database view name	""
ViewKey	string	the database key field (request id)	""
Vendor	bool	TypeSchema is Vendor	""
VendorName	string	Vendor name identifies the plugin supplying the table	""
VendorTable	string	Vendor Table is selected when defining the table to ARS	""
KeyZeroFill	bool	Set true unless the schema has defined max length of the request id field '1' as 1 implying no zero fill.	1
KeyLen	integer	Length of the request id field ('1'), almost always 15 even in those cases where ARS indicates a maximum length of 1 (indicating a maximum length of 15 and no zero fill).	15
KeyPfx	string	The initial value of the request id field. Acts as a prefix to a zero filled integer.	""
KeyPfxLen	integer	Length of the request id field's initial value or prefix	0
StatusNum	integer	Number of Status (field 7) values. A zero indicates that this form has no Status and Status History.fields.	0
ArchEnable	bool	Archiving enabled	0
ArchType	string	One of None, Form, Delete, Form&Delete	None
ArchName	string	The Archive Form Name	""
ArchNoAtt	bool	The "No Attachments" option	0
ArchNoDry	bool	The "No Diary Fields" option	0

Software Tool House Inc

Licensing

Licensing

How It Works

Meta-Update is licensed on a server by server basis.

Licenses are dated and may be indefinite or limited term. Evaluation licenses and migration project licenses are examples of limited term licenses.

Once a Meta-Update license is granted for an ARS Server, Meta-Update scripts can be run against that ARS Server at any time or on any server or workstation.

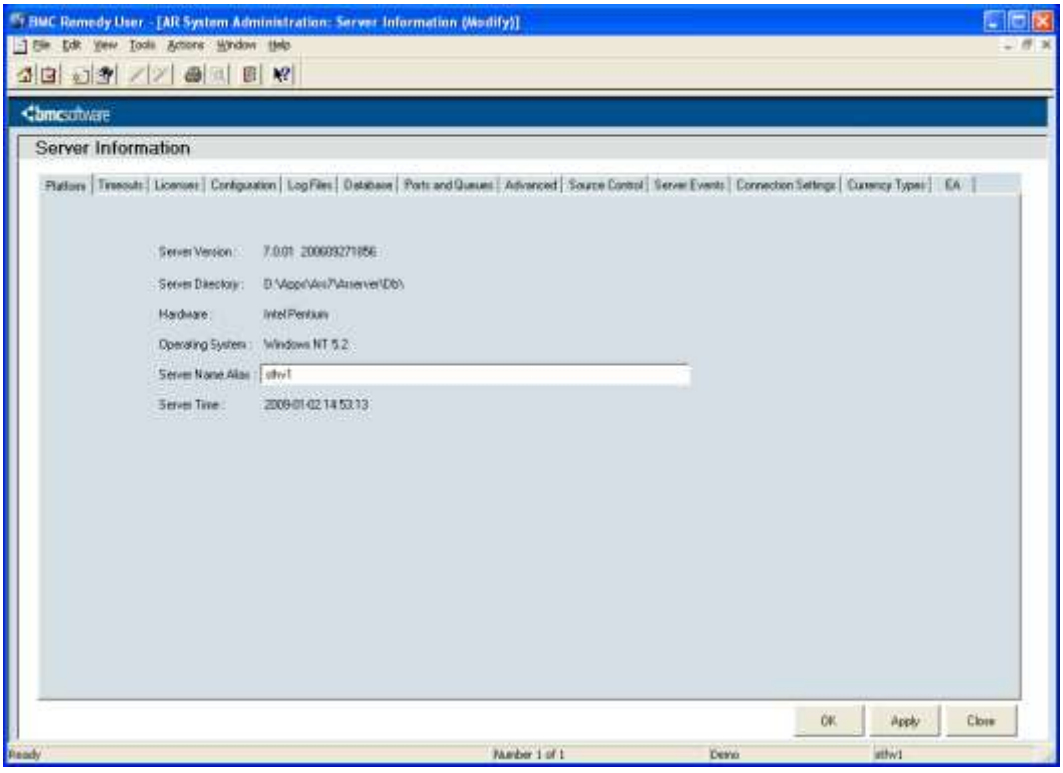
License keys may be requested from the Software Tool House's web site at http://www.softwaretoolhouse.com.

When requesting license keys, the ARS Server Name must be supplied. This ARS Server Alias Name is matched against the supplied license key.

This name is specified in the ARS Server's server configuration file, `ar.conf`, or, `ar.cfg`. It is given by the `Server-Name` value.

On ARS Systems from release 7.0 onward, this setting is available under the user tool when signed on with a User having the Administrator group permission.

From the standard "Home" form, click the "AR System Administration Console" link. From the Administration Console, expand the "System" and "General" branches of the menu, then click the "Server Information" item. Once, the "Server Information" form comes up, click the "Platform" tab.



If an ARS Server is unnamed, the short host name is used as the server name.

Meta-Update may also be licensed on an enterprise-wide basis by matching the ARS Server's domain name. With an Enterprise license, Meta-Update may be run against any ARS server whose host name matches the licensed domain name.

For enterprise licensing, the ARS server's reported full host name is checked against the IP stack and the licensed domain name is matched against the true host domain name.

Stand-alone machines that do not return a domain name cannot be licensed through the site licensing facility.

All licenses carry a term date, support options, the highest release that may be freely upgraded to.

The licensed releases of Meta-Update will run against a licensed server until the term date is reached.

Meta-Update evaluation licenses do not limit Meta-Update in any way. Full functionality is provided in an evaluation license for the term.

Specifying the License Key

The license key can be given to Meta-Update in the following ways:

- In the environment variable, `SthMupdLic=`
- In the script file's `[Main] License=` keyword.
  This can be a reference to an environment variable,
- In a control record on a form on the ARS server

Specifying the License Key with Environment Variables:

On Windows, environment variables may be set on a single DOS session, for a specific user's complete Windows sessions, or for all users' Windows system environment.

To set a single DOS box's environment, open a Command Prompt, then, use the set command to assign the License Key to the expected environment variables. For example,

```
set SthMupdLic=QF143G6-PL95SQ
```

If you do have a site license, there are two more environment variable you may want to set:

`SthSite`	`=`	Site name
`SthDomain`	`=`	Site licence domain suffix

For site licensing, all three environment variables must be defined. For server licensing only the first variable must be defined.

Specifying the License Key in the Script

The Meta-Update License may be specified in the **`[Main]`** section of a script file as an alternative to using environment variables or using a form on the server. To specify the license key in the **`[Main]`** section, code the **`License=`** keyword with the license key as the value.

For site license, you may also code the **`Domain=`** and **`Site=`** values.

`License =` This is the password for either a server or a site license. It must be specified exactly as was specified when the license was requested. If a site license is being specified, both the Site= and Domain= are be required.

`Site =` This is the Site Name the of a site license. It must be specified exactly as was specified when the site license was requested.

`Domain =` This is the Domain suffix for a site license. It must be specified exactly as was specified when the site license was requested.

The following script file addition would accomplish the same thing as the environment variable example above:

```
[Main]
License      =      QF143G6-PL95SQ
```

Specifying the License Key in an ARS form with the User Tool

All Software Tool House tools can reference a special form on the target server for both licensing and operational parameters.

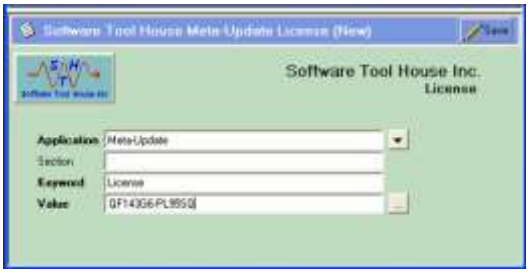
This form needs to be installed using the ARS Remedy Administrator Tool. Simply import all definitions included in the SoftwareToolHouse.def file which comes in the Meta-Update distribution. See below for more information about installing this form.

To add the license key, bring up the SoftwareToolHouse form in the ARS Remedy User Tool. Note that the ARS user must have Administrator privileges to see this form.

If you are replacing a key, search first using “Meta-Update” as the Application and “License” as the keyword. Then modify the Value field to reflect the new license key.

If you are creating a new record, select “Meta-Update” as the Application. Leave the Section field null. Specify “License” for the Keyword and then type in the license key.

For the above example, the form would look like this:

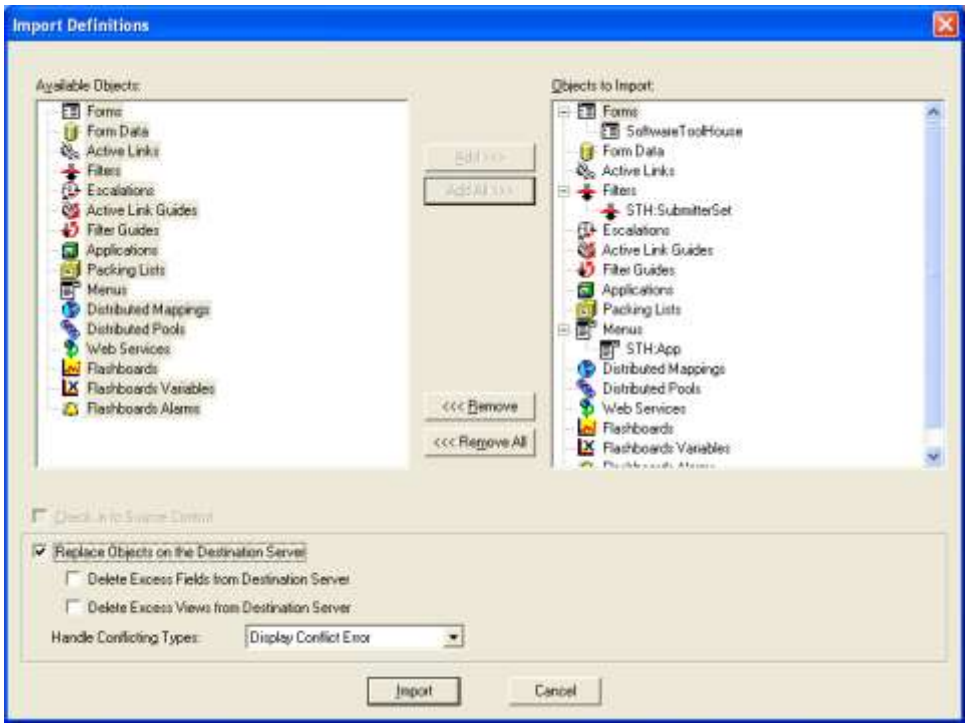


Installing the def File

The License key and other server wide settings for Software Tool House Inc. Applications can be specified in a form called **SoftwareToolHouse**.

An ARS Remedy Administrator Definition file (***`SoftwareToolHouse.def`***) is included in the Meta-Update distribution. You may also download the .def from the web. See http://www.softwaretoolhouse.com/products/ShtMupd/licensing.htm

By running the Remedy ARS Admin tool, you can import this ***`SoftwareToolHouse.def`*** file into your ARS server.



You can control access to Meta-Update by controlling access to the SoftwareToolHouse form.

This form has the following structure:

Application	Text	32
Section	Text	32
Keyword	Text	32
Value	Text	255

Application, Section, and Keyword must be unique. The field Ids are not important.

The License key consists of an uppercase alphanumeric string with hyphens. It is stored in a record of the SoftwareToolHouse for, where

Application	is	“Meta-Update”.
Section	is	$NULL$
Keyword	is	License
Value	is	the license key supplied to you

ARS User Password Encryption

ARS Authentication Password Encryption

ARS user Passwords can be encrypted using a utility. Once encrypted, only the same OS user that encrypted the password can use that password.

The **`SthLic.cmd`** and **`SthLic.sh`** scripts that set environment variables for a licensed server and authentication can be automatically generated on all supported platforms – see below.

These files will allow the setting of environment variables by specifying the desired server, so that for example, a query can be run against one server and then run against another server.

All utilities bundled with Meta-Update will accept either plain text or encrypted passwords in all the methods that ARS Passwords may be set: on the command line, in scripts, or, in environment variables.

If using ARS password encryption, the supplied passwords must be encrypted for each Windows or Unix user that will use Meta-Update. The encryption / decryption is dependent on the currently signed on user.

A new version of the files **`SthLic.cmd`** and **`SthLic.sh`** must be generated for each Windows or Unix user even if the ARS User is the same.

This means that when using ARS Password Encryption, the files, **`SthLic.cmd`** and **`SthLic.sh`** cannot be copied from machine to machine, and, if a single machine is used by more than one user, different **`SthLic.cmd`** and **`SthLic.sh`** files will need to be used by each user.

SthLicUpd Maintenance Utility

This utility can be used to encrypt ARS passwords and to generate an **`SthLic.cmd`** and **`SthLic.sh`** scripts based on license files found in a specified file.

The utility is available on all supported platforms and may be run in prompt mode where it will ask you for all needed information.

Alternate names, ARS Server IPs, Ports, Users, and Passwords can be set. ARS Passwords by, default, are encrypted.

The `SthLicUpd.exe` utility will generate only one of the **`SthLic.cmd`** and **`SthLic.sh`** files as appropriate for the system that it is being run on.

Files produced by SthLicUpd containing encrypted passwords are not transferrable across platforms or users. You have the choice to encrypt the ARS User's password.

Usage

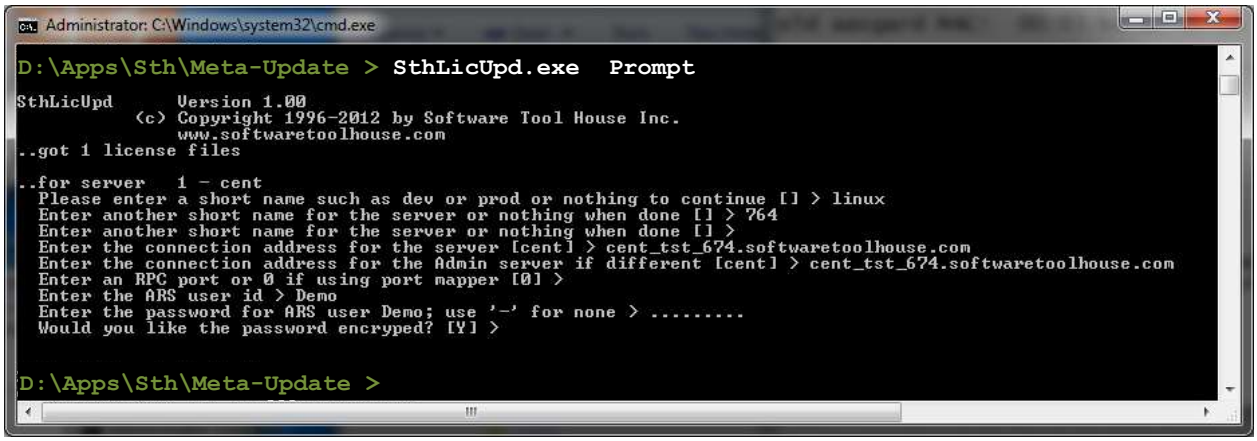
```
Function:
  SthLicUpd is used to modify a Meta-Update SthLic.sh

Modes:
  SthLicUpd can be run in different modes
      Prompt     will scan license files, prompt for needed info
                   and generate a new SthLic.sh.
      Pwd        will encrypt a single ARS user's password or modify
                   your SthLic.cmd file's passwords.

Synopsis:
  SthLicUpd.exe   mode   [ switches ]

    where:
      mode                     is one of:  Prompt or Pwd
           Prompt              Run in interactive mode to scan licenses
                                 can supply -lics and -out arguments
           Pwd                 Will encrypt ARS users' passwords

      switches                 are as follows:
        -licpath   path        The path for the license files;
                               must be a directory
        -out       file        The output path and file name
                   Default for -out is SthMupd.sh in the bin directory
                   that contains SthLicUpd.exe; that bin directory is also
                   the default for finding license files.
      Miscellaneous switches
        -d                     Specifies full tracing

  See: http://www.softwaretoolhouse.com for the Meta-Update User's Guide.
```

Prompt mode will find all available *.lic files in a single directory and ask for any needed information. It will then generate a new `SthLic.cmd` or `SthLic.sh` file.

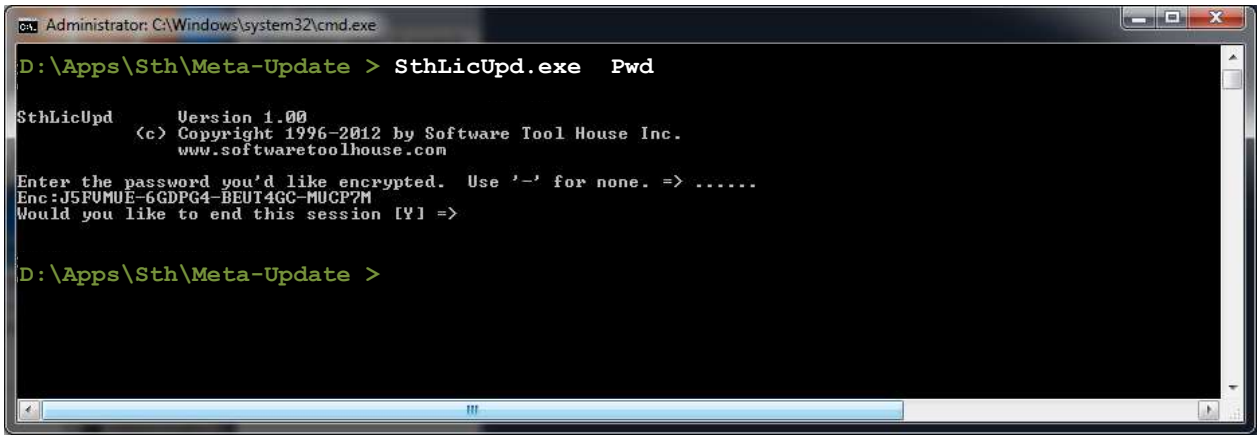
These files will set the environment variable for ARS server connectivity and authentication which all Meta-Update utilities will automatically pick up.

Password mode will simply allow you to encrypt any number of ARS User passwords. You can then use these encrypted password strings in any of the Meta-Update utilities to authenticate to the ARS server.

Sample Prompt Session:



Sample Password Session:



Using the generated `SthLic.cmd` or `SthLic.sh` files

On Unix the generated file must be set to be executable. To do so, enter the following command:

```
> chmod +x ./SthLic.sh
```

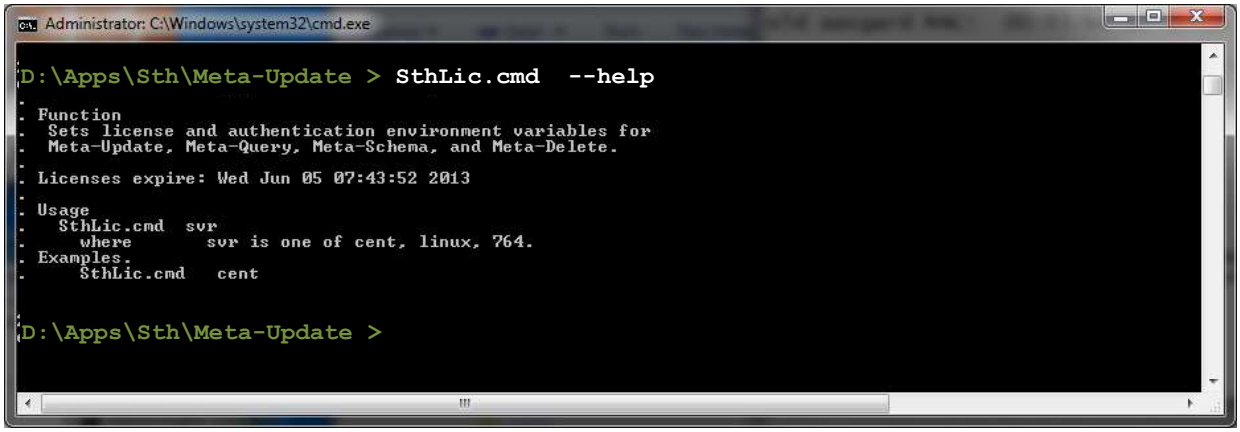
The shell script needs to be "Sourced" or any changes made to environment variables will be lost upon its completion.

When executing the shell script, source it by prefixing the command invocation with a dot, as follows:

```
> . ./SthLic.sh
```

On Windows, simply execute the batch file normally:

```
> .\SthLic.cmd
```



The **`SthLic -help`** command will list all licensed servers and alternate names for the servers.

If the command has already been issued, that is, if the appropriate environment variables are already defined, the command will report the currently set server.

In the above example, a single server is licensed and it has two alternate names given for the convenience of **`SthLic`** users.

Samples

Samples

The following examples can be used as learning vehicles and are included in the distribution package. The distribution may be downloaded from the web.

AR Schema Report

This simple script creates a CSV listing the tables in an ARS server and the number of records they contain. It queries the `arschema` table, does a select count(*) for normal tables, and generates a CSV.

Development time: fifteen minutes!

This is a good beginners' script. It shows how to code a script so that server information is referenced and the script can be used in many ARS server environments without changes.

22 lines!

In addition this demonstrates a SQL Query iteration using `QuerySql=` as well as a `QuerySql=` used inside a LookUp section.

AR Info Report

This simple script creates a CSV listing the predefined server information tag, AR_INFO. It is useful for gathering the server information required for a BMC ticket. The script simply loops through the predefined AR_INFO and outputs a CSV file.

Development time: fifteen minutes!

This is a good beginners' script. It shows how to code a script so that server information is referenced and the script can be used in many ARS server environments without changes.

22 lines!

Ticket Creation Batch Command

A simple script that creates a ticket accepting different command line parameters.

Development time: *one* hour!

This script demonstrates the simple creation of a record based on command line arguments. It introduces the common elements of a Meta-Update script.

Closed Ticket Duplicator

Real Customer Problem

A mail robot must not reopen a ticket, nor attach an email to a closed ticket.

Development time: *three* hours!

This ticket replicator creates a new ticket, with the salient data from the old ticket, assigning it to the last group that closed the old ticket, replicating all emails and other associated records, and finally linking the two tickets together for the GUI button.

This script demonstrates launching other sections so that multiple tables are processed.

Server data extract

Real Customer Problem

Development time: *three* hours!

A single customer has many locations, people, services, etc. This script is used to copy a single customer's data from production to development for a single developer replacing any customer contact information with the developer's information.

This was used in a large development team of a bespoke telecoms client to facilitate development and testing.

Server delta copy

Development time: *one* hour!

A simple script copying all changed records from one server to another – say a read only, reporting server..

Demonstrates using Read Servers, QuerySql, Merge, Query, Update, the Copy assignment command.

AR Schema Report

This simple script outputs a CSV containing the tables defined an ARS server and well as the number of records they have (for regular and join tables) by querying and processing the arschema table.

The script demonstrates:

- How to use environment variables to specify the server
- How to process a QuerySql=
- How to use Output= to create a CSV
- How to code a LookUp returning an SQL function

The script's real function is to demonstrate how you can code the ARS server connectivity arguments so that they are coded as environment variables. This way, if you want to use the same script across several environments, you need only change a few environment variables

Requirements

We don't need to do much in the script itself as we want it to demonstrate how to code the ARS server connectivity and authentication arguments referencing environment variable so that we can run this script easily across multiple ARS Server environments. So, to add a bit of useful function, we'll list the regular tables from arschema along with the current number of records they hold.

Setting Up The Environment

In Windows, the environment variables may be set on a single DOS session, for a specific user's complete Windows sessions, or for all users' Windows system environment.

To set a single DOS box's environment, open a Command Prompt.

Then, use the set command to assign the expected environment variables. For example,

```
set ArsSvr=192.168.1.10
set ArsPort=0
set ArsUsr=Demo
set ArsPwd=-
set SthMupdLic=QF143G6-PL95SQ
```

When you run the script, the script will connect to the ARS server as specified by the above environment variables and validate the above license with that ARS server.

```
[Main]
Server     = $ ENV, ArsSvr  $
User       = $ ENV, ArsUsr  $
Password   = $ ENV, ArsPwd  $
Port       = $ ENV, ArsPort $
```

Script Output

```
Msg: Schema: object_search_admin           Id: 1173    Recs: 1
Msg: Schema: object_search_details         Id: 1174    Recs: 80710
Msg: Schema: object_search_ref             Id: 1175    Recs: 808783
```

Meta-Update solution

Development time:
fifteen minutes!

```
#
#  File:         Rpt-RegTablesRecCounts.ini
#
#  Function:     Test connecting to a server through environment vars.
#                  ArsSvr       server
#                  ArsPort        port      (set to "0" for no server)
#                  ArsUsr         user
#                  ArsPwd         password  (set to "-" for no server)
#                  SthMupdLic   must be set for the above server
#
#  Change History      see bottom of file.
#-------------------------------------------------------------------

[Main]
# [Main] gives the connectivity arguments for the primary,
#        update ARS Server.
# In this case, we code the connectivity info using references
#   of environment variables.  When these variables are not
#   defined, connectivity errors result.
# Use -v on the Meta-Update command to see
#   values used for the ARS Connection.
#
Server     = $ ENV, ArsSvr  $
User       = $ ENV, ArsUsr  $
Password   = $ ENV, ArsPwd  $
Port       = $ ENV, ArsPort $
# License picked up from SthMupdLic= environment
#   variable.  This script coded License= value is over-ridden
#   by the environment variable.  Command line arguments trump
#   environment variables as well.
License    = xxx
# Extra, optional vars
TimeoutShort  =  40
TimeoutNormal =  60
TimeoutLong   = 200
# Client type (can be used by workflow)
ClientType    = 0


[Do]

# This is the entry point and simply issues a
#   QuerySql= on arschema looking for regular (data holding) tables

QuerySql   = ArSch,                                                  &
             @na,                                                    &
             select name, schemaid from arschema                     &
             where schematype = 1
AssignPre  = asg-Msg
```

Set the Server and both connectivity and authentication parameters as references to environment variables.

Single record operations such as Get, Update, Create.

Multi-record operations such as Query and QuerySql.

Admin operations. Not used by Meta-Update.

Script entry point. Issues a QuerySql for all regular forms in arschema

```
[asg-Msg]
#
# Available Tags:
#   ArSch     An SQL record with two columns:
#               Name (1) and SchemaId (2)

# The function of this "Pre" assignment section is to issue the
#   warning message by way of the @Cmd = Msg assignment command.
# Prior to doing that, we need to issue an SQL query to get the
#   number of records in the form.

# This LookUp assignment is passed the ARS Table Schema ID and
# uses a select count(*) query on the "T" table.
@Cmd       = Ref, V, Count,            &
                  @LookUp,             &
                  LkUp-Count,          &
                  $ArSch, 2$
#


# This is the real output of the script:
#   a message to the console
@Cmd  = Msg, W, Schema: $ArSch, 1$ \t\t\t     &
                Id:     $ArSch, 2$ \t\t\t     &
                Recs:   $V, Count$



[LkUp-Count]



# We are passed the arschema schema id and
#   simply do a select count(*) on that schema id's "T" table
#
QuerySql        = ArSchCnt,                                    &
                  @na,                                         &
                  select count(*) from T$CTL, LookUp_Src$
QuerySqlTarget  = $ArSchCnt, 1$
```

All work (issuing a message) is done in this non-targeted assignment section. There is no output.

This ***Assignment Command*** **sets a script variable - $V, Count$ - to the single SQL row column:**

```
select count(*)
from Txxx
```

This ***Assignment Command*** **issues a message to the console. It uses the ArSch SQL"record" and the "V" -record - $V, Count$ of script variables.**

This ***LookUp Section*** **returns a single string: the value of the single SQL column from the single SQL row:**

```
select count(*)
from Txxx
```

AR Server Info Report

This simple script outputs a CSV containing the fields and values of the predefined AR_INFO Tag. This Tag is automatically defined for every Meta-Update script and is the ARS Server Information.

The script demonstrates:

- How to use environment variables to specify the server
- How to process a fields Loop=
- How to use Output= to create a CSV

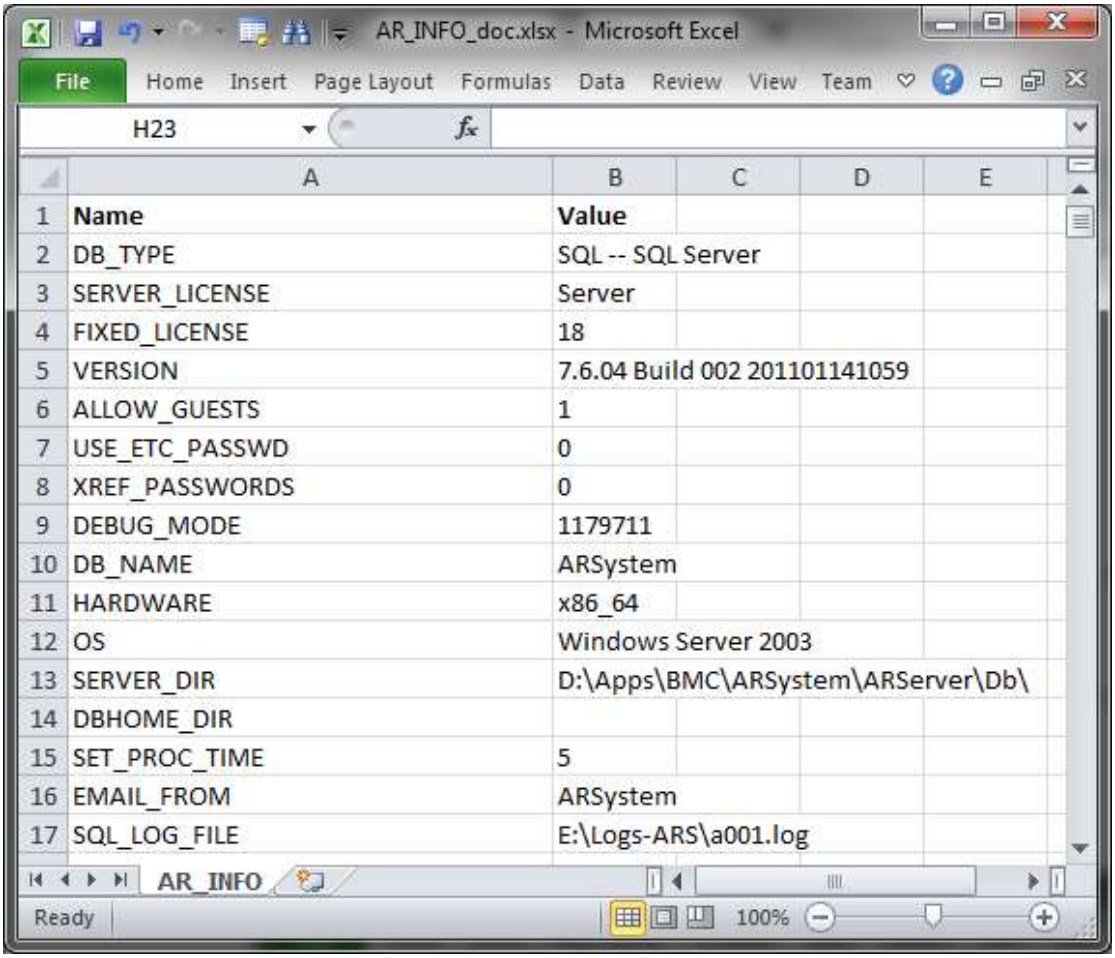
Requirements

There are no requirements for this script and no server performance ramifications.

Setting Up The Environment

This script uses the default environment variables for connectivity as set by **`SthLic.cmd`** or **`SthLic.sh`**.

Script Output



	A	B
1	**Name**	**Value**
2	DB_TYPE	SQL -- SQL Server
3	SERVER_LICENSE	Server
4	FIXED_LICENSE	18
5	VERSION	7.6.04 Build 002 201101141059
6	ALLOW_GUESTS	1
7	USE_ETC_PASSWD	0
8	XREF_PASSWORDS	0
9	DEBUG_MODE	1179711
10	DB_NAME	ARSystem
11	HARDWARE	x86_64
12	OS	Windows Server 2003
13	SERVER_DIR	D:\Apps\BMC\ARSystem\ARServer\Db\
14	DBHOME_DIR	
15	SET_PROC_TIME	5
16	EMAIL_FROM	ARSystem
17	SQL_LOG_FILE	E:\Logs-ARS\a001.log

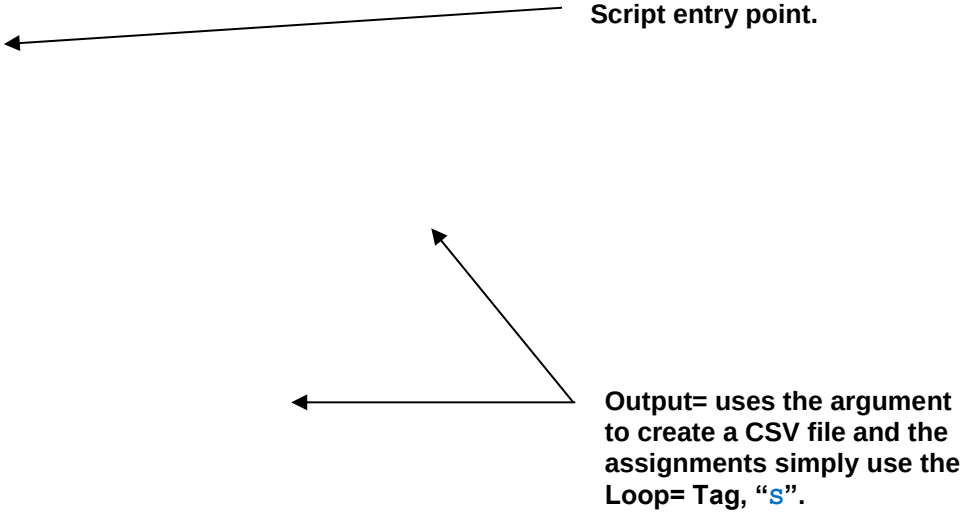
Meta-Update solution

Development time:
fifteen minutes!

```
#-------------------------------------------------------------------------
#  Meta-Update is copyright (c) 1996-2012 by Software Tool House Inc.
#                           www.softwaretoolhouse.com
#-----------------
#
#  This is a Meta-Update script that simply writes the automatic
#    AR_INFO values to the screen.
#
#  It simply loops through the fields of the automatic Tag, AR_INFO,
#    and outputs a CSV row for each to the passed CSV file.
#
#-------------------------------------------------------------------------
[Main]
#  This [Main] section gives script arguments and server info.
#
ArgNm     = outf

PrmReq  = 1, . Function:
PrmReq  = . This is a Meta-Update script that simply writes AR_INFO
PrmReq  = .   values from the Server to a CSV file in the form
PrmReq  = .     Name           Value
PrmReq  = .     DB_TYPE         SQL -- SQL Server
PrmReq  = .     VERSION         7.6.04 Build 002 201101141059
PrmReq  = .
PrmReq  = . Usage
PrmReq  = .   SthMupd   $CTL, ScriptFx$  Do  -p   out-file
PrmReq  = .     where       out-file    is the output file CSV
PrmReq  = .

[Do]
#  This is the "main entry point" of the script.
#  It iterates through all the "fields" of the AR_INFO tag
#    - an automatically defined tag containing all Server info -
#  and for each, ouputs a CSV row in the specified file.
Loop      = Fields,   S,    AR_INFO
Output    = F,        Fle,  $Arg, outf$
Assign    = Do-asg

[Do-asg]
#  Each field value pair is output
#    into a single CSV record
Name      =  S,    FieldName
Value     =  S,    Value

[Fle]
#  This defines the CSV file with two fields
#Type     = Delimited, ",", FldHdr
Fields    = Fle-Flds
Format    = Csv

[Fle-Flds]
Name      = $
Value     = $
```

A single argument is required: the name of the output file.

The Loop= iterates through all the fields of AR_INFO assigning a set of attributes such as FieldName and Value to the Tag, "S"


Script entry point.
Output= uses the argument to create a CSV file and the assignments simply use the Loop= Tag, "s".

Software Tool House Inc

Ticket Creation Batch Command

This is an invented script built as an example to help learn Meta-Update. The script is untested and it must be noted that the script will need editing before being run in any reader's environment.

Requirements

We need a simple, easy to use, parameterized, ticket generator for our ARS Help Desk. We want to be able to create new tickets so that we can, if desired, force an assignment to a specific group.

We want to use this callable command in various ways:

- Remedy ARS workflow and escalations,
- Scheduled jobs through "at" or "cron",
- Configured commands in other their network monitors
- Added as a last step of some of their bespoke software

The command would depend on the arguments given. Defaults would be assumed for all null arguments.

- Requester Email or Requester login
  If it contained an "@" it would be looked up in a people form as an email. Otherwise it would be looked up as a login name.
- Subject — The subject of the ticket.
- Description — The full textual description.
- Category — If not supplied, use "Default"
- Type
- Item
- Assignment Group — Only assign if supplied.

Meta-Update solution

Development time: *one* hour!

```
[Main]
Server          =       Sth2
User            =       Demo
ArgNm           =       Subject
ArgNm           =       ReqSearch
ArgNm           =       Description
ArgNm           =       Category
ArgNm           =       Type
ArgNm           =       Item
ArgNm           =       AsgGrp

PrmReq          =       2

[TT-New]
#Simply create a Ticket every time.
Schema              =   HPD:HelpDesk
Assign              =   Asg-New-TT

[Asg-New-TT]
Subject         =   Arg, Subject
Description     =   Arg, Description

@Cmd                =   @if("$Arg, ReqSearch$ ==")
  LoadQ             =   Req,                                    &
                        SHR:People,                             &
                        'Login'    =   "Default Requester")
@Cmd                =   else
  @Cmd              =   @if("$Arg, ReqSearch$ ~="@")
    LoadQ           =   Req,                                    &
                        SHR:People,                             &
                        'Email'    =   "Default Requester")
  @Cmd              =   else
    LoadQ           =   Req,                                    &
                        SHR:People,                             &
                        'Login'    =   "Default Requester")
  @Cmd              =   endif
@Cmd                =   endif

Requester Id        =   Req, 1
Requester Login     =   Req, Login

Category    =   @if("$Arg, Category$ == "",                     &
                    "Default",                                  &
                    "$Arg, Category$")
Type        =   @if("$Arg, Type$ == "",                         &
                    "Default",                                  &
                    "$Arg, Type$")
Item        =   @if("$Arg, Item$ == "",                         &
                    "Default",                                  &
                    "$Arg, Item$")

Assignment Group =  @if("$Arg, AsgGrp$" != "")                  &
                    Arg, AsgGrp
```

Names the arguments

Specifies that only 2 arguments are required and usage info when not enough arguments supplied.

Simple command section that always creates a single record in the HelpDesk schema

Simple assignment of passed argument value

Load the requester record into memory under the tag, *Req*

Assignment of data from loaded Requester record.

Assign either "Default" or the supplied values.

Only make this assignment if a value was supplied.

The Category, Type, and Item assignments are simply based on the passed arguments on an individual basis. To make similar assignments on a hierarchical basis, simply use this segment instead or the three individual Category, Type, Item assignments above:

```
@Cmd      =   @if("$Arg, Category$ == "")
  Category    =   "Default"
@Cmd          =   else
  Category    =   Arg, Category
  @Cmd        =   @if("$Arg, Type$ == "")
    Type      =   "Default"
  @Cmd        =   else
    Type      =   Arg, Type
    @Cmd      =   @if("$Arg, Item$ == "")
      Item    =   "Item"
    @Cmd      =   else
      Item    =   Arg, Item
    @Cmd      =   endif
  @Cmd    =   endif
@Cmd      =   endif
```

The PrmReq can be used to specify usage information as well as the required number of arguments. The usage information is delivered when an insufficient number of arguments is supplied on the command line. Note that passing a null value – "" – is still passing a value. Named arguments not supplied on the command line contain the null value.

This example is equivalent to the above but will supply usage information when used incorrectly.

```
PrmReq    =   4, Usage:
PrmReq    =   .    TT-New -p  Subj, Desc, Req,  Cat, Typ, Item, AsgGrp
PrmReq    =   .  where
PrmReq    =         Subj      is required and is the ticket short subject
PrmReq    =         Desc      is required and is the long
PrmReq    =         Req       is either the requester login or email
address
PrmReq    =                   Default Requester assumed if null
PrmReq    =         Cat       Category (Default if null)
PrmReq    =         Typ       Type     (Default if null)
PrmReq    =         Itm       Item     (Default if null)
PrmReq    =         AsgGrp    is an assignment group or null
PrmReq    =   .  Create a ticket and optionally assigns it to a group
```

Closed Ticket Replicator

This is taken from a customer solution. It has been modified to be used as a Meta-Update sample. The script demonstrates how to launch other dependent command sections, how to make assignments from multiple records, how to use the Copy assignment command.

Background

The customer had a series of Perl scripts to control ticket generation and filing emails with tickets. This allowed a full email conversation between the ticket agent and ARS system and the requester.

Sometimes a requester would reply to an email after it was closed. The customer's business process stated no further work could be done on a closed ticket.

As such, a mechanism would be needed to create a new ticket from the old ticket selecting work history records and emails.

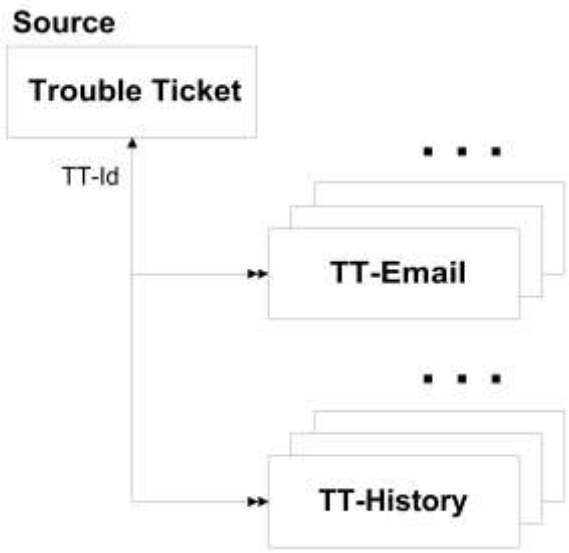
Requirements

A Perl callable ticket replicator was needed. It would create a new, open, assigned ticket, containing the emails, the work history with a few extra generated records identifying the email to the closed ticket. It would copy pertinent data from the old ticket.

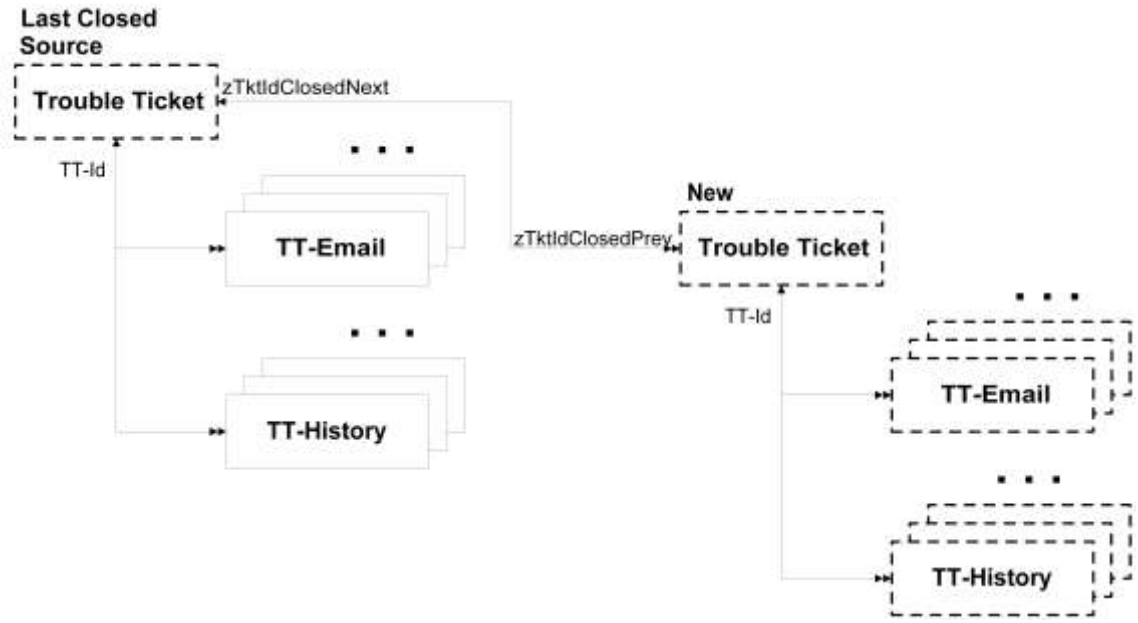
The new ticket would be created assigned to the resolving group of the closed ticket.

The two tickets would be linked for a GUI facility to allow ticket chains to be followed. The closed ticket would need to be updated with the new ticket's id.

This image shows the schemas and records of a single ticket.



The dashed lines in this image show the desired updated and created records:



Meta-Update solution

``

Development time: *three* hours!

```
[Main]
Server          =       Sth2
User            =       Demo
ArgNm           =       TtIdClosed
ArgNm           =       IdLog
PrmReq          =       2

[TT-Copy]
# this section creates a Ticket every time.
Schema          =   TT-TroubleTicket
LoadQ           =   Src_TT,                                    &
                    TT-TroubleTicket,                          &
                    '1' = "$Arg, TT-IdClosed$"
Create          =   New_TT,                                    &
                    TT-TroubleTicket
Merge           =   Yes
Assign          =   asg-TT-New
Launch          =   TT-Orig-Upd,
Launch          =   TT-Email
Launch          =   TT-Hist-1, TT-Hist-2,                      &
                    TT-Hist-3, TT-Hist-4,                      &
                    TT-Hist-5
Launch          =   TT-New-Upd

[asg-TT-New]

Status          = New
zTktIdClosed    = Src_TT, 1
zTktIdClosedNew = $NULL$
@Cmd            = @if("$Src_TT, Next Action$" != "")
  Next Action     = "Old closed ticket actions:\n"
  Next Action     = "==========================\n"
@Cmd            = endif
Next Action     = Src_TT, Next Action
Ticket Type     = Problem
Priority        = Medium
Severity        = 4
Ticket Opened   = $DATE$
Ticket Closed   = $NULL$
Problem Started = $NULL$
Problem Fixed   = $NULL$
Escalate when   = $NULL$
# The next cmd copies all non-assigned fields.
@Cmd            = Copy, Src_TT, DupIgnore, CoreAssign, Skip: 1
```

Names two required arguments.

The closed source TT is loaded into memory from the passed Id.

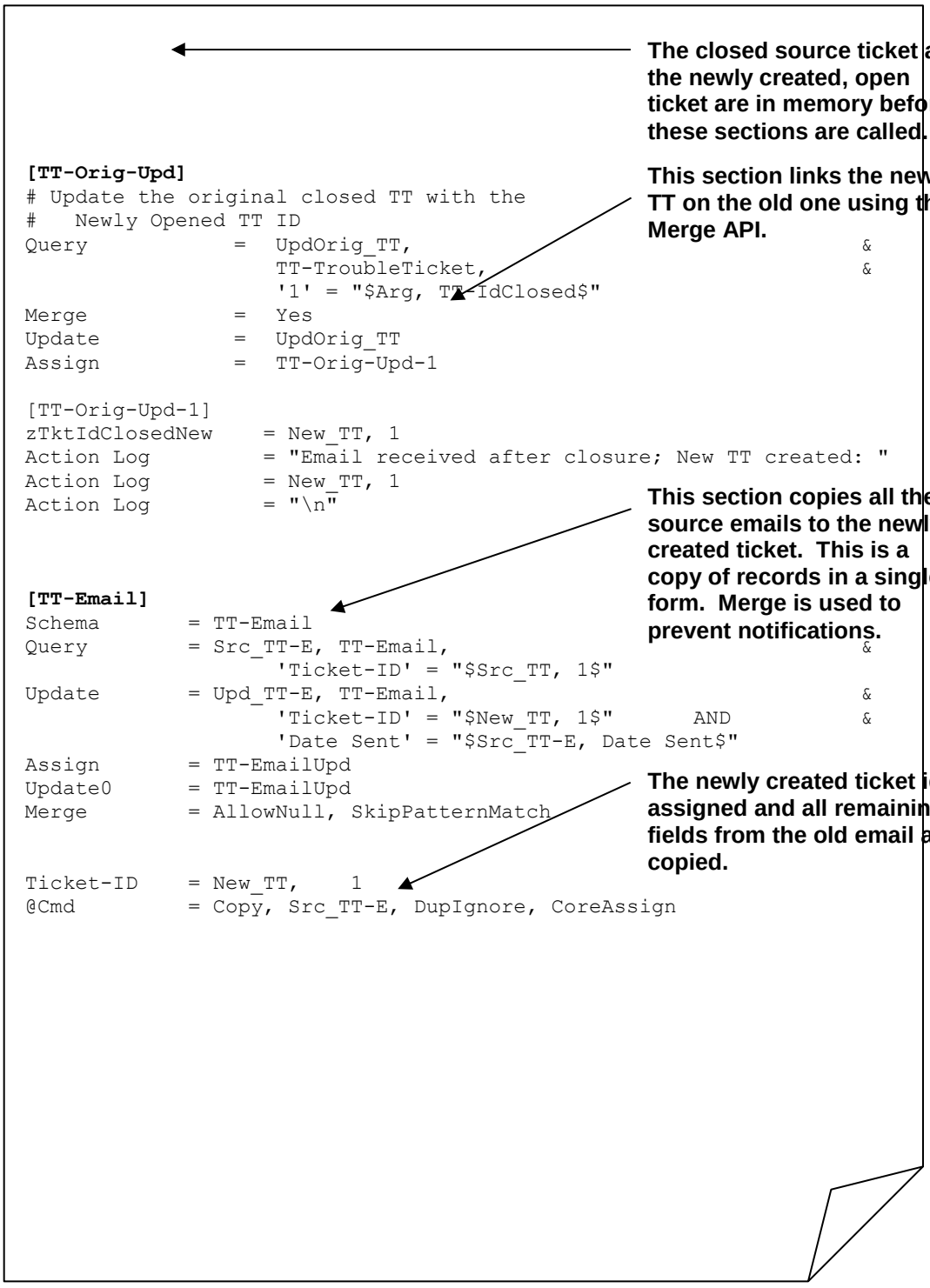
Always creates one single record in the HelpDesk schema

Merge API is used to inhibit Submit filters.

The created record is loaded into memory after submission and other sections are run to copy dependent records.

Assignments for the new TT some arbitrary values (constants) and the remaining set of fields from the closed ticket

Meta-Update solution




```
[TT-Orig-Upd]
# Update the original closed TT with the
#   Newly Opened TT ID
Query            =  UpdOrig_TT,                                  &
                    TT-TroubleTicket,                            &
                    '1' = "$Arg, TT-IdClosed$"
Merge            =  Yes
Update           =  UpdOrig_TT
Assign           =  TT-Orig-Upd-1

[TT-Orig-Upd-1]
zTktIdClosedNew    = New_TT, 1
Action Log         = "Email received after closure; New TT created: "
Action Log         = New_TT, 1
Action Log         = "\n"


[TT-Email]
Schema       = TT-Email
Query        = Src_TT-E, TT-Email,                                 &
                    'Ticket-ID' = "$Src_TT, 1$"
Update       = Upd_TT-E, TT-Email,                                 &
                    'Ticket-ID' = "$New_TT, 1$"      AND          &
                    'Date Sent' = "$Src_TT-E, Date Sent$"
Assign       = TT-EmailUpd
Update0      = TT-EmailUpd
Merge        = AllowNull, SkipPatternMatch


Ticket-ID    = New_TT,    1
@Cmd         = Copy, Src_TT-E, DupIgnore, CoreAssign
```

The Main section

The `PrmReq=` specifies that three arguments are required. A better one might be:

```
PrmReq  = 3, TT-Closed-Copy.ini copies a closed TT to a new, unassigned TT
PrmReq  = .
PrmReq  = .  usage
PrmReq  = .    SthMupd.exe  TT-Closed-Copy.ini TT-Copy  -p  TT-ID-src  IdLog
PrmReq  = .
PrmReq  = .  where
PrmReq  = .    TT-ID-src    Parm 1    the closed ticket's ID that will be copied
PrmReq  = .    IdLog        Parm 2    the file name for the IdLog
PrmReq  = .
PrmReq  = .  function
PrmReq  = .    Will create a new TT as a copy of the old one including all its
PrmReq  = .      previous emails but not its history records for which a few will
PrmReq  = .      be artificially generated.
PrmReq  = .    Will also update the source TT with the newly generated ID plus
PrmReq  = .      a text reference to the generation...
PrmReq  = .    The source TT must not have already been copied to a new TT.
PrmReq  = .
```

TT-Copy, The Called Command Section.

The command section called to copy a ticket is: `TT-Copy`.

To call the command, either on the command line or within a shell script or batch file, one could enter:

```
SthMupd.exe  ./TT-Cpy.mus  TT-Copy  -p  TKT000049  TKT000049  /tmp/..
```

`TT-Copy` has no `Query=`, `QuerySql=`, `File=` so it is executed exactly once.

The `Load=` keyword loads the closed source ticket. The data of this ticket can be referenced with `$Src_TT, field$`. This can be used in subsequent queries or assignments.

The `Create=` keyword causes an ARS record to be submitted. This could have been an Update= keyword which would have allowed different assignments for an update or a create operation. An ARS query that selects exactly one or zero update records must be specified.

It loads the source ticket record which is always the last ticket closed in a chain of tickets. That id is passed on the command line as the named argument, `TT-Closed`.

After the command section creates the new ticket, that new ticket is re-read so that all fields have the current values, and the launches are processed in order.

Launching Other Command Sections.

Each launch allows a new command section to be processed. That command process has all the preceding sections' references available to it. It can query and iterate like any other section.

```
Launch          =  TT-Orig-Upd,
Launch          =  TT-Email
Launch          =  TT-Hist-1, TT-Hist-2,                              &
                   TT-Hist-3, TT-Hist-4, TT-Hist-5
Launch          =  TT-New-Upd
```

Command Section	Overview
`TT-Copy`	The called or main section. It executes only once

	and creates a new ticket. It then launches, in order, these other sections.
`TT-Orig-Upd`	Uses a Merge to add the new ticket id reference to the old ticket.
`TT-Email`	Uses a Query= to copy all emails to the new ticket.
`TT-Hist`	Uses a Query= to copy all the history records.
`TT-Hist-1, 2, ..5`	Uses a Create= to create a few new history records for the TT-Closed-Copy operation.
`TT-New-Upd`	

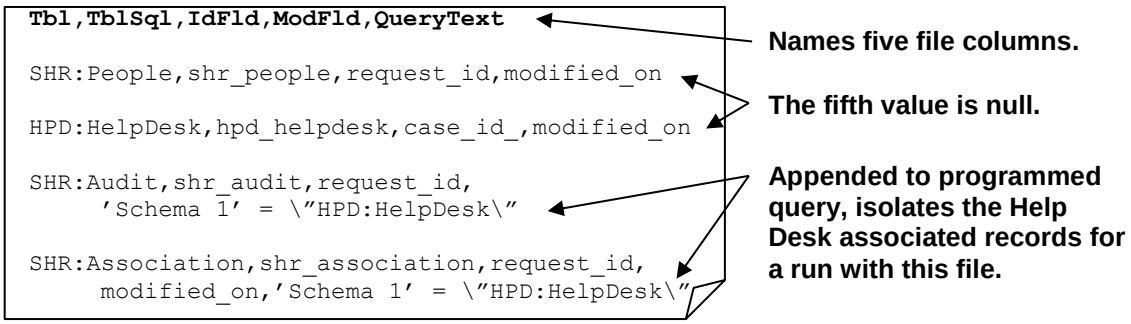
Server Delta Copy

This script is created as a learning vehicle to demonstrate several Meta-Update statements.

Requirements

A reporting server must be kept in sync with a production server. The sync job is run on a 24 hour delay basis. The updated records are to be transferred based on the last modification date. Request IDs are to be maintained. The subset of the tables to be kept synchronised is given by an ASCII file. That file also specifies query text that can be appended to the programmed modification date query.

The following is a sample file



Interestingly, multiple jobs can be simultaneously to take advantage of the ARS server's multi-threading. This could be extended to several machines. Each job would specify independent sets of dependent tables.

Script Overview

The Main section will define the source server. It will also change the date into a format suitable for an SQL query.

The called command section will process the passed CSV file. It will not make any outputs itself, but instead, launch another command section.

That launched section, will in turn issue an SQL Query on the table named in the CSV and a date with any optional query text appended.

That query section will actually do an SQL query to prevent ARS timeouts as generally the modified by field is not indexed. It will iterate through that list updating any records it needs to.

This Script Demonstrates

- Processing a CSV with a File=.
- Using an assignment section to prepare a query string.
- Using an assignment section to convert a date from a normal format to an integer for an SQL query.
- Using a Read Server. In a LoadQ and a QuerySql.
- Specifying an Update query.
- Using the Copy assignment command.
- Using a Launch.

Meta-Update script

``

```
[Main]
Server          =       Dev01
User            =       Demo
ReadServers     =       Main-Prod
ArgNm           =       inp-csv-fle
ArgNm           =       mod-date
AArgNm          =       idlog
PrmReq          =       3
IdLog           =       $Arg, idlog$.log

[Main-Prod]
Tag             =       Prod
Server          =       Dev02-prod-copy
User            =       Demo
Port            =       3201

[Fle-Tbl]
Type            =       Delimited, ",",FldHdr
Format          =       Excel
Fields          =       Fle-Tbls-Flds

[Fle-Tbl-Flds]
Tbl             =       $       # table name in ARS
TblSql          =       $       # table name in SQL view
IdFld           =       $       # '1' in SQL
ModFld          =       $       # '8' in SQL
QueryText       =       $       # SQL query text

[SvrSync-Date]
# Processes the passed CSV file of tables to synchronise.
File            =   Ftbls,                                          &
                    Fle-Tbl,          ,                             &
                    $Arg, inp-csv-fle$"
AssignPre       =   asg-Mk-Qry
Launch          =   Tbl-Sync

[asg-Mk-Qry]
# will append an "and" and any extra query text
#   supplied in the CSV row
@Cmd = Ref, Vars, Qry
       $Ftbls, ModFld$ > $Arg, mod-date$
@Cmd = @if("$Ftbls, QueryText$" != "")                              &
      Ref, Vars, Qry   $Vars,  Qry$ AND ( $Ftbls, QueryText$ )
```

Specifies the script's tag, ip and login for the production server.

Names three arguments.

All arguments are required.

Declares the format and field name for the passed CSV file.

The file's first record contains the field names which must match these fields.

This is the called section. It iterates through the file's rows.

The `AssignPre=` section is run after the next file record is loaded but before any Launches are processed.

This makes an "and .." string if the CSV had an optional QueryText value.

```
[Tbl-Sync]
# Issues an SQL query to obtain the modified
#   record IDs, Loads the records and updates
#   them on the target server.

QuerySql      =   @Prod,                                    &
                  SqlLst,                                   &
                  @na,                                      &
                  select $Ftbls, IdField$                   &
                  from   $Ftbls, TblSql $                   &
                  where  $Vars, Qry$

LoadQ         =   @Prod,                                    &
                  Src,                                      &
                  $Ftbls, Tbl$,                             &
                  '1'=   "$SqlLst, 1$"

Update        =   Tgt,                                      &
                  $Ftbls, Tbl$,                             &
                  '1'=   "$SqlLst, 1$"
Merge         =   Yes, NoWorkflow
Assign        =   asg-Copy
AssignNew     =   asg-Copy

[asg-Copy]
@Cmd          = Copy, Src, CoreAssign
```

This section has the CSV row loaded and does the rest of the work by issuing the SQL Query on the source server for the modified request ids, loading the record, and updating the record on the target server.

This section copies the source record's fields including core fields.

Script Detail

The `[Main]` section does these things:

1. Specifies three argument names with the `ArgNm=` keyword.
2. Specifies the file to be generated as the id log with the `IdLog=` keyword..
3. Says that all three arguments are required but does not give additional user help text when those arguments are not specified on the command line.
4. Establishes the server and authentication parameters for the update server
5. Establish the server and authentication parameters to the source server through the `ReadServers=` keyword. The value of that keyword is a section name which, like the Main section gives server and authentication parameters for addition servers. Note the `Tag=` keyword in the `[Main-Prod]` section. Queries will use this tag - `@Prod` - to reference the addition server.

The Called Command Section

The `[SvrSync-Date]` section is specified on the command line and is the script "entry-point".

The `File=` keyword says we will iterate through a columnar file. The `[Fle-Tbl]` section specifies the attributes and fields of the file. Row one of the file contains the field names and must match the fields specified in the CSV.

The `AssignPre=` allows us to build the select SQL query for the modified date using the fields as specified in the file row and the optional query text also specified in the file row.

The first assignment of `[asg-Mk-Qry]` makes the modification date query text for the SQL statement using the modification field name specified in the CSV file for this table and the time argument passed on the command line. This is set in tag “Vars”, field “Qry”.

If the CSV query text was non-null, the same string is appended with “and (..)” using the supplied query text.

Now that the SQL query string has been made, the section launches the actual worker section `[Tbl-Sync]` to copy the modified records. This section has no output.

The Launched Section

Section `[Tbl-Sync]` is launched once for each table / row in the passed configuration file row. That row is in memory when this launched section is invoked. In addition, a select Query string has been created.

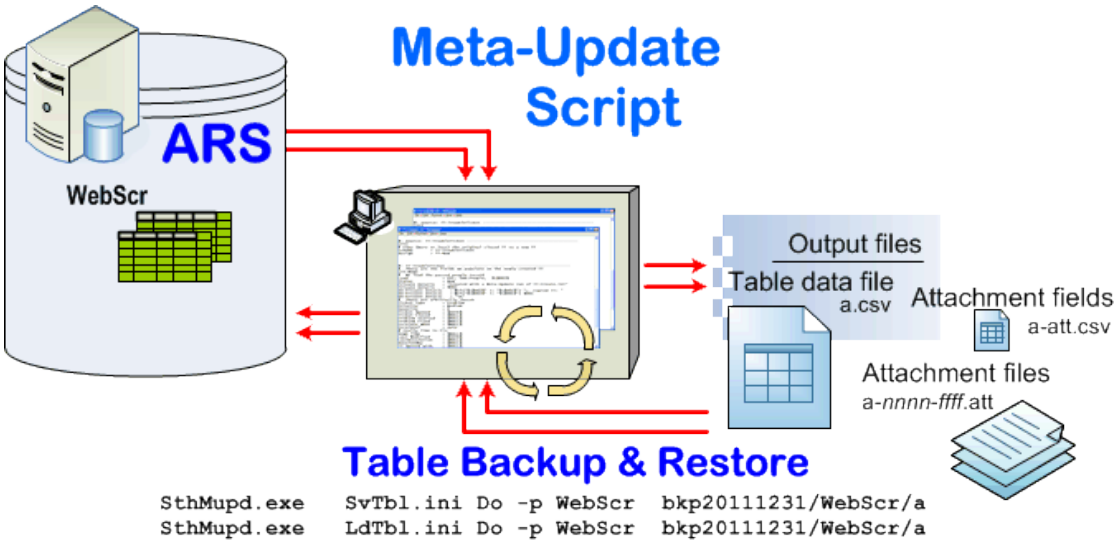
This section issues a select to retrieve the ids of the modified records for the given table. It does this with the `QuerySql=` keyword, specifying the @Prod server tag. The @na says that we will not name or edit any of the columns returned by the select statement, instead referring to them by their column numbers.

We iterate through the set of Request Ids returned by the select. During each iteration, we load the source record from the source server with the `LoadQ=` keyword, and issue the `Update=` to create the same record on the target server with the same request id as in the source server. That Update or Create is performed using the Merge API and no filters are fired – including filters set to fire or Megre.

The `Assign=` and `AssignNew=` sections are the same and simply issue the Copy command to copy all source fields including attachments and core fields into the target record, updating or creating that record,

ARS Table Backup and Restore

There are two scripts in this sample, one to back up a table and the other to restore a table.



To back up any ARS table, run the SvTbl.ini script passing as arguments, the table name, and a backup file prefix. The restore script will take as input the same table name and same file prefix.

The backup script will generate these files:

- a single csv containing all data from each field of the passed table
- if and only if there are attachment fields in that table, a CSV of the field names and field ids for these attachment fields
- a file prefixed by the passed prefix for each attachment.

The restore script will process these files as a set:

- a single csv containing all data from each field of the passed table
- if the attachment fields CSV exists, will read these attachment fields and ids into a script array
- if there are attachment fields, and the data CSV indicates a non-null attachment, a file saved by the backup script will update the attachment content and have the original attachment name.

This script introduces more complex features of Meta-Update. The script demonstrates:

- Query=, Output=
- Field Loops
- Output files based on schemas
- Schemas and Queries passed as arguments
- extracting and loading attachments

Running the script.

The package is in the distribution and may also be downloaded from the script library.

The package contains a def file for the form _Test. It also contains data saved by the sample save script that can be used to populate the _Test table.

To validate these scripts, simply run the backup against a single record, generate a report of all data from this record, delete the record, run the restore, generate a second report from this

record, and, do a difference of the two reports. There should be no differences between the two reports.

```
SthMupd  SvTbl.ini Do -p _Test test "'1' = \"000000000000001\""
SthMqry  -f -S _Test "'1' = \"000000000000001\"" > rpt-before.txt
SthMdel  _Test  "'1' = \"000000000000001\""
SthMupd  LdTbl.ini Do -p _Test test
SthMqry  -f -S _Test "'1' = \"000000000000001\"" > rpt-after.txt
diff     rpt-before.txt   rpt-after.txt
```

Backup Script Overview

`[Do]` is the main command section and issues the query against the passed table. Each record is assigned to the tag `Src`.

An AssignInit is used to initialize script variables and formulate a default query string (1=1) if the script was not passed a query qualification.

`[Do]` will output a record to a CSV for each record it processes. It will not change any values other than encoding any embedded quotes and line feeds. The assignments to the output CSV are handled by a single copy command. The file's fields are also copied from the passed table name.

`[Do] will Launch [Sv-Att-Struct]` once only.

> `[Sv-Att-Struct]` creates a second CSV containing a list of all the Attachment fields Field Names and IDs).
>
> If there are no attachment fields, the CSV is not created. The single Launch is controlled by the script variable `$V, First$` which is initialized to TRUE and set to FALSE by an AssignInit in `[Sv-Att-Struct]`.
>
> If there are any attachment fields, the CSV is created and a variable is set to indicate that there are attachments that should be saved.

`[Do]` will Launch `[Sv-Att]` each record it processes if there are any attachment fields in the table. This is controlled by the `$V, gotAtt$` script variable which was set by `[Sv-Att-Struct]`

> `[Sv-Att]` iterates through all non-null attachment fields in the `Src` record. So, for any single record it may iterate zero or more times.
>
> `[Sv-Att]` has no record or file output, so all work is done in an AssignPre section which is called after the Loop's Tag is assigned on each iteration.
>
> The assignment is a simple AttachmentSave command issued to save the attachment to the file system. The file is named as follows:
>
> `-prefix- ReqId - FieldId .att`
>
> Prefix is passed on the command line, ReqId is the request id field with any '|' characters (from Join forms) translated to '-'. This is done through a simple regular expression used for the side effect of allowing a Subst field specification.

Meta-Update script

```
# Meta-Update sample script file.
# Meta-Update is copyright 1996-2011 by Software Tool House Inc.
#
# File:                    SvTbl.ini
#                          Part of the sample scripts for Meta-Update.
#
#        Two scripts used to save and restore any ARS tables' data.
#        This is the Save script.  See LdTbl.ini for the restore script
#
#        This Save script will save all records into a single CSV
#        and attachments into files prefixed by the passed argument.
#-------------------------------------------------------------------



[Main]
#  The main section gives sign-on info and declares
#    Script arguments required and usage info.

Server   = $ ENV, ArsSvr  $
Port     = $ ENV, ArsPort $
User     = $ ENV, ArsUsr  $
Password = $ ENV, ArsPwd  $

PrmReq      = 2,. Function
PrmReq      =  .  Two scripts used to save and restore ARS tables.
PrmReq      =  .  This is the Save script.
PrmReq      = .
PrmReq      . Usage:
PrmReq      .   SvTbl.ini  Do -p   tbl  outp  [ qry ]

ArgNm       = schema
ArgNm       = F-out
ArgNm       = qry

#-------------------------------------------------------------.do
[Do]
#
# This is the main entry point and called routine. This section
#   reads through the given table creating the output CSV file
#
# A Query is executed on the source table and the output file record
#   record is created using an assignment copy command.
#
# Once only, a section that saves a CSV of attachment files is
#   launched.  If there are attachment fields, a section is
#   launched each record to save those attachments to the file system.
#
```

Server connectivity and authentication set from environment variables.

```
#[Do]
#
AssignInit  = asg-I
Query       = Src,                                              &
              $Arg, schema$,                                    &
              $V,   Qual$
Output      = Tgt,                                              &
              Out-f,                                            &
              $Arg,  F-out$.csv
Assign      = asg
Launch      = @if("$V, First$")  Sv-Att-Struct
Launch      = @if("$V, gotAtt$") Sv-Att


[Out-f]
#
# This declares the output CSV file.
#
Type        = Delimited, ",", FldHdr
Format      = Quoted always Quotes escape lf escape
Fields      = Out-f-flds

[Out-f-flds]
@Cmd        = Copy, $Arg, schema$

[asg-I]
#
#  This "initial" assignment section initialises script variables
#  Input Tags
#    Arg  Ptn           "" or a query string
#  Output Tags
#    V    First         do Attachment File output one time
#    V    gotAtt        table has attachments; set Sv-Att-Struct
#    V    Qual          "1=1" or the passed query string
#    V    AttPth        the attachment path
#
@Cmd        = Ref,   V,  gotAtt,    0
@Cmd        = Ref,   V,  First,     1
@Cmd        = Ref,   V,  Qual,      "1=1"
@Cmd        = Ref,   V,  AttPth,    "$Arg, F-out$"
@Cmd        = @if("$Arg, qry$" != "")                           &
              Ref,   V,  Qual,  "$Arg, qry$"
[asg]
#
# This is the assignment to the CSV file.  Because all fields
#   from the table and CSV file match, we just issue a copy
#
@Cmd        = Copy,  Src
```

The ARS Schema is a reference. As is the Query qualification.

The output file name is the passed prefix appended with ".csv"

The ARS Schema's fields are copied into the output file's definition.

This single command assigns all fields from the table to the CSV converting embedded line feeds and quotes as specified.

```
[Sv-Att-Struct]
#
#  This section saves the field names and ids of any attachment fields
#    into a special CSV processed by the companion script.
#
#  Input Tags
#    Src                  The source record
#  Output Tags
#    V    First           0    we want to execute once only
#    V    gotAtt          1    says we have attachment fields
#
Loop        = Fields, Att,  Src, Type Attachment
Output      = TgtS,                                                    &
              Out-f-struct,                                            &
              $Arg,  F-out$.att.csv
Assign      = Sv-Att-Struct-asg
AssignInit  = Sv-Att-Struct-asg-Init


[AssignInit  = Sv-Att-Struct-asg-Init]
@Cmd       = Ref,   V,  First,     0

[Sv-Att-Struct-asg]
@Cmd       = Ref,   V,  gotAtt,    1
AttFldNm   = Att,  FieldName
AttFldId   = Att,  FieldId



[Out-f-struct]
#
# This declares the output CSV file listing the attachment fields
#
Type       = Delimited, ",", FldHdr
Format     = Quoted always Quotes escape lf escape
Fields     = Out-f-struct-flds

[Out-f-struct-flds]
AttFldNm   = $
AttFldId   = $
```

If there are no attachment fields, the loop is executed zero times, no file is created, and `gotAtt` is not set true.

No matter if there are any attachment fields or not, we want to set `First` false.

```
[Sv-Att]
#
#  This section extracts any Attachment fields into the file system
#  Input Tags
#    Src                   The source record
#  Output Tags
#    Att                   The @info for each attachment field
#
#  The AssignInit simply gets rid of any '|' in the request id value.
#
Loop       = Fields, Att,  Src,
             Type Attachment, NoNulls
AssignPre  = Sv-Att-asg


[Sv-Att-asg]
#
#  Here we are processing all non-null attachments in the record
#  We save them to the file system using the name:
#      id1-id2-fid.att
#    where id1   is the request id (with Join forms' | changed
#      to hyphens)
#    and   fid   is the attachment field id
#


#  An easy way to change '|' to - is by a Subst; we match
#    the whole string for the Subst to be effected.
@Cmd       = Ref,  V,   Sv-Att-asg-regex,                          &
             @regex,  /(.*)/,  $Src, 1$

#  Now extract the attachment under the new file name which the
#    companion script will expect for non-null attachments.
@Cmd       = AttachSave, Src,  $Att, FieldName$,                   &
             $V, AttPth$-$V, ReqId$ $Att, FieldId$.att


[Sv-Att-asg-regex]
#
#  This field list is for the @regex that is used to change '|'
#
ReqId      = $ Subst /|/-/
```

This will loop through all attachment fields with non-null values in the record just loaded.

There is no output; an AssignPre is called after the next iteration is loaded and this saves the attachment.

We use a regex that always matches to effect a Subst. This results in $V, ReqId$ holding a request id with all '|' changed to '-'.

This saves the attachment to the file system under a unique name.

Restore Script Overview

`[Do]` is the main command section and does no iteration or output instead only Launching two sections once.

An AssignInit is used to initialize script variables. There is no Query argument in the restore script. The AssignInit also determines if an Attachment Fields CSV exists or not. It does this with a Reference spawn assignment that assigns "OK" to the stdout variable if the file exists.

Note that because of the UNIX if shell syntax the stdout and stderr redirection does not come at the end of the command line and is explicitly stated.

`[Do] will Launch [Do-Att-Flds]` once only.

`[Do-Att-Flds]` processes the Attachment Fields CSV just building a "script array" of Attachment Field Names and Field IDs and setting the number of attachment fields.

If there are no attachment fields, the CSV was not created and the number of attachment fields remains 0.

`[Do-Att-Flds]`makes no output, so only an AssignPre is used. That AssignPre section increments the number of attachment fields counter and sets the Field Name and Id into the array.

```
[Do-Att-Flds-asg]
#
#  For each field, increase the number of fields,
#    and set it in the Va, Fnm and Fid arrays
#
@Cmd       = Ref,   Va, Max, @eval, $Va, Max$+1
@Cmd       = Ref,   Va, @,   Do-asg-FF

[Do-asg-FF]
Fnm$Va, Max$  = F, AttFldNm
Fid$Va, Max$  = F, AttFldId
```

Increment `Va, Max`

This assigns a series of "field / value" pairs to the `Va` tag. We use references in the fields to be assigned.

Tags built are like this:

```
Va,    Max       2
Va,    Fnm1          Attachment1
Va,    Fid1          5378001021
Va,    Fnm2          Attachment2
Va,    Fid2          5378001022
```

`[Do]` then Launches `[Do-Load]`, the backup file handling section, since all Attachment fields are now known.

`[Do-Load]` Processes the passed backup data file and updates the passed table using '1' = the first field of the file" as the update query.

Like the backup script, the File's fields are copied from the schema and the schema in the query and the file's field's copy is the `$Arg, schema$` reference.

Because the file's fields are copied, the file's field 1 is the first schema field, or field '1', and this is used in the Update= query.

The Update is done with the Merge API and with Merge workflow inhibited. It is only through the Merge API that core fields may be set (such as Request ID, Submitter, Create Date).

Note that this restore script will not work with Join forms unless Merge workflow is allowed. A write to a join can only write to the database if the filters on that join fire.

The Assignment section for the ARS Table Update= is the same for new or updated records.

If there are any attachment fields and the backup data indicates that it is non-null, a string is assigned with two file names:

```
original attachment name, attachment file

C:\dir\xxx.xxx, -prefix-  ReqId – FieldId .att
```

Meta-Update can process attachment values as references, single file strings, or double file strings. In the case of a double file string, the second string is the file in the file system that contains the data of the attachment, and the first name is the file name set into the attachment value.

Because the file is copied from the table, a simple copy assignment command will set all fields to their backed up values skipping any fields that have already been assigned a value.

Meta-Update script

```
# Meta-Update sample script file.
# Meta-Update is copyright 1996-2011 by Software Tool House Inc.
#
# File:                    LdTbl.ini
#                          Part of the sample scripts for Meta-Update.
#
#        Two scripts used to save and restore any ARS tables' data.
#        This is the Load script.  See SvTbl.ini for the backup script.
#
#        This Load script will process the CSV files generated by the
#        save script and load all records including any attachments
#-------------------------------------------------------------------

[Main]
#  [Main] gives sign-on info and declares Script arguments.
Server    = $ ENV, ArsSvr  $
PrmReq      =  .   LdTbl.ini  Do -p   tbl  outp

ArgNm       = schema
ArgNm       = F-inp
#-------------------------------------------------------------.do
[Do]
#
# Before we can proceed with loading the data file, we'll need a list
#   of Attachment fields so that we can assign them as needed.
#
# So, here, the AssignInit figures out if the attachment fields CSV
#   exists, then, launches [Do-Att-Flds] to save attachment fields in
#   script variables, and finally launch the Do-Load section to process
#   the backup data file against the ARS table.
#
AssignInit  = asg-I
Launch      = @if("$Va, Do$") Do-Att-Flds
Launch      = Do-Load


[asg-I]
#
#  This "initial" assignment section sets $Va, Do$ to the existence of
#    the "$Arg, F-inp$.att.csv" file and makes a few initializations.

#  Input Tags
#    Arg    F-inp       the output file name
#  Output Tags
#    Va     Max        init num attachment fields to 0
#    Va     Do         set to true if file $Arg, F-inp$.att.csv exists.
#
@Cmd        = Ref,   Va, Max,        0
@Cmd        = Ref,   Va, Do,         0
@Cmd        = @if("$CTL, OS$" == "UNIX")
  @Cmd        = Ref,   V,  @spawn,                                  &
      if [ -f '$Arg, F-inp$.att.csv' ] ;                            &
      then echo OK $redir$ ;                                        &
      fi;
@Cmd        = else
  @Cmd        = Ref,   V,  @spawn,                                  &
      if exist "$Arg, F-inp$.att.csv" echo OK
@Cmd        = endif
@Cmd        = @if("$V, stdout$" ~= "OK")
  @Cmd        = Ref,   Va, Do,         1
@Cmd        = endif
```

The AssignInit section `[asg-I]` sets `$Va, Do$` to true if the Attachment Fields CSV exists in the expected location.

Note different command to determine file existence n Windows and Unix. Note use of `$redir$` in Unix command.

The echo produces "`OK<lf>`" or "`OK<cr><lf>`" in `$V, stdout$`, so we just check for a leading OK.

```
 [Do-Att-Flds]
#
#  The SvTbl companion script generated an attachment fields CSV.
#    We are only Launched if this file exists!
#  We set number of attachment fields for the Update= assignments.
#
#  Input Tags
#    Va   Max              0        number of attachment fields
#  Output Tags
#    Va   Max              0 + n    number of attachment fields
#    Va   Fnm1,2, ..       char     field name array 1..n
#    Va   Fid1,2, ..       int      field id   array 1..n
#
File        = F,                                                    &
              Inp-f-att,                                            &
              $Arg,  F-inp$.att.csv
AssignPre   = Do-Att-Flds-asg

[Do-Att-Flds-asg]
#
#  For each field, increase the number of fields, and set it in the
#    Va, Fnm and Fid arrays
#
@Cmd       = Ref,   Va, Max,       @eval,  $Va, Max$+1
@Cmd       = Ref,   Va, @,         Do-asg-FF

[Do-asg-FF]
Fnm$Va, Max$  = F, AttFldNm
Fid$Va, Max$  = F, AttFldId


# File declarations:  the two input CSV files
#   Inp-f-att       saved by SvTbl.ini; schema's attachment fields

[Inp-f-att]
#
Type       = Delimited, ",", FldHdr
Format     = Quoted always Quotes escape lf escape
Fields     = Inp-f-att-flds

[Inp-f-att-flds]
AttFldNm   = $
AttFldId   = $
```

Increment `Va, Max`

This assigns a series of "field / value" pairs to the `Va` tag. We use references in the fields to be assigned to build an array.

```
[Do-Load]
#
# Loops through the given CSV (created by the companion script)
#   updating in the target table with the value of the first CSV
#   field (Request ID) being matched against '1'
#
# We need to use Merge (like the Import Tool) so that we can assign
#   core fields like '1' etc. For Joins, remove NoWorkflow from Merge.
#
#  We know the number of attachment fields, their names, and ids, so
#    if the attachment fields are non-null, they are assigned with
#    their original file name and the expected file system name.
#
#  The remaining field values are simply copied from the CSV row.
#
File         = Src,
               Inp-f,
               $Arg,  F-inp$.csv
Update       = Tgt,
               $Arg, schema$,
               '1' = "$Src, 1$"
AssignNew    = Do-Load-asg
Assign       = Do-Load-asg
Merge        = Yes, NoWorkflow


[Do-Load-asg]
#
# This is the assignment to the ARS record from the CSV file
#   (with the same fields as the ARS record).  Because all fields
#   from the table and CSV file match, we just issue a copy.
#
# We need the CoreAssign option because we want '1', '2', etc assigned
from the CSV - only available with Merge
#
#  If the attachment value in the CSV is non-null, we will have a
#    file named:   id1-id2-fid.att
#     id1 etc    is the request id (with '|' changed to hyphens)
#     fid        is the attachment field id

#  We change '|' to - with a Subst; we match all for the Subst
@Cmd       = Ref,  V,   Ld-Att-asg-regex,  @regex,  /(.*)/,  $Src, 1$



[Ld-Att-asg-regex]
#
#  This field list is for @regex used to substitute hyphens for '|'
#
ReqId      = $ Subst /|/-/
```

We use the reference $$src, 1$ to indicate the first CSV field which will be Request ID, Entry ID, and so on.

You cannot use NoWorkflow on Join forms.

```
[Do-Load-asg]

# handle attachments separately
@Cmd       = @if("$Va, Max$")
  @Cmd       = Ref,  V,  @info,   Src,  $Va, Fnm1$
  @Cmd       = @if("$V, Value$")
    @Cmd       = Ref,  V,  attval,                                    &
           "$V,  Value$,$V, AttPth$-$V, ReqId$-$Va, Fid1$.att"
    $V, FieldName$ =  V, attval
  @Cmd       = endif
  @Cmd       = @if("$Va, Max$" > 1)
    @Cmd       = Ref,  V,  @info,   Src,  $Va, Fnm2$
    @Cmd       = @if("$V, Value$")
      @Cmd       = Ref,  V,  attval,                                  &
                 "$V,  Value$,$V, AttPth$-$V, ReqId$-$Va, Fid2$.att"
      $V, FieldName$ =  V, attval
    @Cmd       = endif
    @Cmd       = @if("$Va, Max$" > 2)
      @Cmd       = Ref,  V,  @info,   Src,  $Va, Fnm3$
      @Cmd       = @if("$V, Value$")
        @Cmd       = Ref,  V,  attval,                                &
                   "$V,  Value$,$V, AttPth$-$V, ReqId$-$Va, Fid3$.att"
        $V, FieldName$ =  V, attval
      @Cmd       = endif
      @Cmd       = @if("$Va, Max$" > 3)
```

The maximum number of attachment fields in any one form should be handled here, with, perhaps, an error thrown if it is exceeded.

```
      @Cmd       = endif
    @Cmd       = endif
  @Cmd       = endif
@Cmd       = endif


@Cmd       = Copy,  Src,   CoreAssign
```

The remaining assignments are handled with a Copy command.

Index

Software Tool House Inc

Index

@

@Cmd
Assignment Commands 169
Field Sections..... 147
@date
Assignment Command..... 186
@eval
Assignment Commands 187
@fmt
Assignment Command..... 185
@include
directive..... 96
Including Script Files 98
List Files Debugging Command 88
@info
Assignment Commands 183
@val
Assignment Commands 185

A

Abort
Assignment Command..... 174
API
Meta-Update API Versions..... 56
Archive Forms
Set Schema Assignment Commands..... 196
Arguments
Meta-Update Usage 65, 66
Arithmetic Expressions
Assignment Commands 187
Functions..... 194
Named Constants..... 194
Operators 193
Random Number function 195
Random Number Seeding..... 195
Using in Assignment Commands 193
ARS records
Assignment Commands 190
Assignment
Commands
AttachSave 174
Sections in Concepts..... 33
Set Schema Command 196
Trace Command 197
Assignment Commands 169
@eval..... 187
Abort..... 174
Arithmetic Expressions..... 187
Arithmetic Expressions, Using..... 193
ARS records..... 190
Client Processes 190
Conditional Assignments..... 187
Copy..... 170
Targets 170
Date Information 186
Double References..... 183
Double References..... 185
Format values..... 185
Include..... 173
Message 175
Reference 176
Variables..... 180
Regular Expressions..... 189, 191
Server Processes 188
Spawn..... 175
Tag Equivalence..... 188
Assignment Sections
Update..... 133
assignments
Concepts 28
Assignments
Concepts 21
`AssignNew=`
Samples..... 277
AttachSave
Assignment Command 174
Auditing
IdLog..... 139

B

BackTrace
Debugging,Command..... 88
Breakpoints
About..... 86
Command in Script..... 83
Setting while Debugging..... 91

C

Cache
LookUps
Keywords..... 213
Client Processes
Assignment Commands..... 190
Command Prompt
Ideal Properties..... 70
Commands
Set Schema Assignments..... 196
Trace Assignment Command 197
Concepts
Assignments Sections 33
Control Sections 33
Create..... 41
Debugging 83
Examples..... 43
Flowchart in Concepts 35
Iteration..... 33
Launch..... 34, 43
Output..... 33, 40
Output Files 42

Update 41
Conditions
Launch 138
Reference Assignment Commands 187
Continue
Debugging Command 90
Control
Sections
in Concepts 33
Control Section
Create in Concepts 41
Examples in Concepts 43
Flowchart in Concepts 35
Launch in Concepts 43
Operational Statementst 111
Output Files in Concepts 42
Output in Concepts 40
Update in Concepts 41
Control Sections
Concepts 20
Keywords 109
Operational Statements 109
Statements 109
Control Statements
Output 135
Query 118
Until 131
Update 133
UpdateIfEqual 134
Copy
Assignment Command 170
Core Fields 173
CoreAssign 173
NoCoreAssign 173
Targets 170

D

date
Assignment Command 186
Debugging
About 83
Backtrace Command 88
Break Command 91
Breakpoints About 86
Commands 87
Continue Command 90
Line Numbers About 85
List Command 87
List Files Command 88
Next Command 89
Print Command 89
Prompt 84
Quit Command 90
Delimiters ..... *See* Loop Statement
Double anchor ..... Loop Statement
Developing
Scripts 77
Distribution
Contents 49
Expanding 48

E

Environment
Run Time 56
Running Meta-Update 55
Environment Variables 60
SthMupdLic 61
SthScriptPath 60
expressions
Conditional assignments 166
Copy assignment command 172
Include assignment command 173
Output statement 135
SQL fields 122
Until statement 109
While statement 125

F

Fields
Copying from ARS Forms 147
Dates and Date Formats 150
Format Specs 148
Formats in @fmt Assignment Command 185
Numbers and Numeric Formats 152
Quotes in values 152
Regular Expression Extracts 189
Sections 145
SQL Selects 149
File
Iteration in Concepts 38
Log Format 75
Logging Locally 72, 73
Trace Format 75
Tracing Locally 72, 73
Format
Assignment Command 185
Functions
in Arithmetic Expressions 194

I

IdLog 139
Ifs
Reference Assignment Commands 187
Include
Assignment Command 173
Including Script Files
@include 98
Installing 46
Distribution Contents 49
Expanding the Distribution 48
Iteration
About 113
Concepts 20
File in Concepts 38
in Concepts 33, 36
Loop
Defined in Concepts 39
Loop in Concepts 39
Query 118

Concepts 37
QuerySql
Concepts 38
Defined in Concepts 38
Types 33
Until 131
Defined in Concepts 40

J

Join
Loop Statement, Example 5 130

K

Keywords
in Control Sections 109

L

Launch
in Concepts 43
Launch
Concepts 21
in Concepts 34
Launch
Conditions 138
License
Meta-Update License Key 59
Line Numbers 85
List
Debugging,Command 87
List Files
Debugging,Command 88
Load statement 117
Loads
defined in Concepts 31
Logging
ARS Client Log Switches 71
Local Log File 65
Local Tracing 72
Message Format 75
Server Tracing 73
Switch Settings 70
The –d Switch 71
Tracing Locally 72
LookUps
About 198, 204
Automatic Tags 200
Caching 213
File 205
Keywords 201
Query 209
QuerySql 210
Reference Assignment Commands 187
Types 199
Loop
Iteration in Concepts 39
Loop Statement
Defined in Concepts 39
Delimiters 126
Example 1 Diary: 128
Example 2 String: 128
Example 3 Fields: 128
Example 4 String: 129
Example 5 Join 130
Examples 128
Sort 126

M

Message
Assignment Command 175
Meta-Update
BMC API Versions 56
Msg
Assignment Command ..... Assignment Commands:Msg
Multifile
Output Statement 135

N

Name
Constants in Arithmetic Expressions 194
Next
Step a Script in Debugger 89

O

Operational Statements
in Control Sections 111
Output
Files in Concepts 42
Output
Concepts 21
in Concepts 33
Output
Files
defined in Concepts 42
Output
Program Output 68
Output Statement
Multifile 135
Multiple CSV files 135
Output Statement 135

P

PCRE
Assignment Commands 189
Positional Arguments
Meta-Update Usage 66
Print
Debugging Command 89
Program
Meta-Update License Key 59
Meta-Update Output 68
Meta-Update Versions 58
Program Arguments
Meta-Update Usage 65, 66

Q

Query 118
  Iteration
    Concepts 37
  Limits 120
  Logging 120
  Performance 120
  QueryMax 121
  QueryStart 121
  SQL ..... See QuerySql
  using field names 37
QueryMax
  999,999,999 121
  with Query 121
QuerySql
  Defined in Concepts 38
  Iteration
    Concepts 38
  LookUps 210
QuerySql statement
  Select field variable 122
QueryStart
  with Query 121
Quit
  Debugging Command 90

R

Random Numbers
  in Reference Assignment Commands 195
  Seeding for Arithmetic Expressions 195
Ref
  Assignment Commands 176
Reference
  Assignment Commands 176
References
  Concepts 22
  defined in Concepts 28
  Simple 28
  String 29
References, String
  defined 30
Regular Expressions
  Assignment Commands 189
Return Values 67
Run Time Environment 56
Running 55
  API Versions 56
  ARS Client Tracing 71
  Environment for Meta-Update 55
  Firing from Workflow 77
  Local Tracing 72
  Log File 65
  Log Format 75
  Logging 70, 71
  Logging ARS Client 71
  Logging Locally 72
  Logging Server 73
  Meta-Update Arguments 65, 66
  Meta-Update Environment Variables .. 60, 61
  Program Output 68
  Return Values 67
  Server Tracing 73
  stdout & stderr 67
  Tracing 70
  Tracing Format 75
  Tracing Locally 72
  Tracing Server 73

S

Script
  Debugging Commands 87
  Source
    Including Files 98
  Source Format 96
Scripts
  Developing 77
  SthMupdLic Environment Variable 61
  SthScriptPath Environment Variable 60
Sections
  Control
    in Concepts 33
  Types in Concepts 32
Select field variable ..... *See* QyerySql Statement
Server Processes
  Assignment Commands: 188
Session Timeouts ..... See Timeouts
Set Schema
  Assignment Command 196
Simple References
  defined in Concepts 28
Sleep Statement 138
Sort ..... *See* Loop Statement, *See* Query statement
Spawn
  Assignment Command 175
SQL
  LookUps 210
  Query
    Concepts 38
Statements
  in Control Sections 109
  Operational in Control sections 111
stderr
  Assignment Commands 190
  Running 67
stdout
  Assignment Commands 190
  Running 67
Step
  Step a Script in Debugger 89
SthMupdLic Environment Variable 61
SthScriptPath Environment Variable 60
String
  References 29
String Reference ... *See* References, String

T

Timeouts
Long ... 103
Normal ... 103
Trace
Assignment Command ... 197
TraceTrace Assignment Commands ... 197
Tracing
ARS Client Log switches ... 71
Local Log File ... 65
Local Tracing ... 72
Message Format ... 75
Server Tracing ... 73
Switch Settings ... 70
Type
Sections in Concepts ... 32

U

Until statement ... 131
Defined in Concepts ... 40
Update
About - Concepts ... 41
Update Statement ... 133
Assignment Sections ... 133
UpdateIfEqual Statement ... 134

V

Value interpretations
Concepts ... 22
Versions
Meta-Update Program Versions ... 58

W

Workflow
Running Meta-Update from ... 77