Two Application Infrastructure Tools (Automated Email and Non-Volatile Locks)

Andrew Ratcliffe

Ratcliffe Technical Services Limited

1 Abstract

The use of SAS software as an industrial-strength application development environment continues to grow. As the size of application suites grows, the need for “infrastructure” systems occurs too. This paper describes two such systems that this author has successfully implemented in the past.

Running regular overnight batch jobs (such as data warehouse loads) requires some kind of error checking. At many sites this comprises one or more staff reviewing the SAS logs each morning to look for errors and warnings. This approach is, at best, tedious and, realistically, unreliable.

This paper presents a solution to the perennial problem by detailing an approach for automatically checking the log and emailing the findings to a selected group of recipients.

In all applications, the need to prevent corruption of data is paramount; in data-entry and data-editing systems, data can get corrupted if the application does not prevent two or more users from updating the same data element at the same time.

On occasions, the facilities provided by SAS software are not flexible or powerful enough for the application in question. This paper provides a design for a SAS/AF class that serialises and controls access to nominated resources within the application environment. Unlike the facilities provided by base SAS software and SAS/SHARE, the SAS/AF class permits the “locking” of resources between SAS sessions, i.e. non-volatile locks. Thus, a user can lock a resource (such as a workflow item or a budget plan) for an unlimited period of time, regardless of whether the user is actively editing the data at any particular point in time.

2 Email

2.1 Introduction

Checking SAS logs after automated runs is a tedious and error-prone business. This section of the paper presents a solution to the perennial problem by outlining an approach to automatically checking the log and emailing the findings to a selected group of recipients.

This approach has two distinct areas: checking the log, and emailing the findings to an appropriate group of people. The first can be simplified to looking for ERROR: and WARNING:, the second can be controlled by a simple data set acting as a control table (with possible completion codes and the appropriate list of email recipients).

2.2 A Brief Analysis

The fundamental requirement is to be able to run an automated (scheduled and unattended) data load process and issue an email upon completion. The email should document the success or failure of the process.

Data loads usually consist primarily of base SAS code. The email should attach the log and provide a summary of it in the body of the email. The recipients of the email should be variable, dependent upon the success or failure of the process.

The success of the load should be determined from the log by default. However, the calling programmer should have the ability to pass his or her own deduction. The default method for determining success or failure should simply scan the log for WARNING: and ERROR: messages and deduce failure if any are found.

The email should contain a brief summary of the SAS log, i.e. a list of ERROR and WARNING messages. The application programmer should be able to issue informational messages to the log and these should be shown in the summary too.

2.3 A Brief Design

The solution has a two-layer approach. Firstly, the basic ability to send an email is encapsulated into a class interface. The class interface provides methods that match with the simple tasks of sending an email. Secondly, a simple method-based interface is introduced that uses a control table to indicate the desired email recipients for (un)successful tasks.

The class interface for basic email is not entirely necessary since using email from the SAS software is relatively straightforward anyway. The approach was taken, however, in order to simplify the use of email to such a degree that the remainder of the coding (the AUTOMAIL methods) could be entirely abstracted from the mundane activities surrounding creating an email.

2.3.1 Example

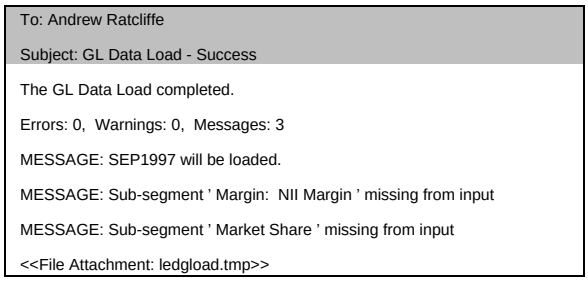
The following example runs a data load process and uses the SEND_MSG method to send an email message to inform recipients of successful or failed completion.

```
/* DIRECT THE LOG TO AN EXTERNAL FILE */
SUBMIT CONTINUE;
proc printto new log='c:\temp\ledgload.tmp';
run;
ENDSUBMIT;

  :
Data load process is coded here
  :

/* RESET STORAGE LOCATION FOR LOG */
SUBMIT CONTINUE;
proc printto;
run;
ENDSUBMIT;

/* SEND A COMPLETION MESSAGE */
call method('utility.utility.automail','send_msg'
           ,'GL','Data Load','c:\temp\ledgload.tmp');

/* DELETE THE EXTERNAL LOG FILE */
rc=filename('temp','c:\temp\ledgload.tmp');
rc=fdelete('temp');
rc=filename('temp','');
```

The list of recipients of the email message will be determined from the AUTOMAIL control table. If any error or warning messages are found, the task will be considered as a failure. The SEND_MSG method will

expect to find either a 'Fail' or a 'Success' record in the AUTOMAIL control table for the given application and event.

The figure below shows an example email message that could result from a successful execution of the example code.



2.3.2 Method Interface

2.3.2.1 AUTOMAIL

The design calls for two methods. These are summarised below:

SEND_MSG

Sends an email message

CALL METHOD ('UTILITY.UTILITY.AUTOMAIL', 'SEND_MSG', *appl, event, logfile <, compcode, summtext, err, warn, msg>*);

Where...	Is type...	And...
appl	C	specifies the name of the application
event	C	specifies the name of the event
logfile	C	specifies the name of the log file
compcode	C	specifies the event's completion code
summtext	N	specifies the identifier of an SCL list containing text to be included in the body of the email message
err	N	specifies the number of error messages
warn	N	specifies the number of warning messages
msg	N	specifies the number of informational messages

Details

If *compcode* is omitted (or blank), the log file will be scanned and the *summtext*, *err*, *warn*, *msg* will be deduced by calling the RVW_LOG method.

The log file must be an external file (not a SAS catalog entry).

The values used for *appl* and *event* must be stored in the AUTOMAIL control table. If *compcode* is supplied, its value must also be stored in the control table with *appl* and *event*. If *compcode* is not supplied, the control table must contain records with COMPCODE values of 'Success' and 'Fail'.

The data stored in the AUTOMAIL control table is case-sensitive.

RVW_LOG

Review the log

CALL METHOD ('UTILITY.UTILITY.AUTOMAIL', 'RVW_LOG', *logfile, logtype, compcode, summtext, err, warn, msg*);

Where...	Is type...	And...
logfile	C	Specifies the name of the log file
logtype	C	Specifies the type of the log file:
		'FILE' log file is an external file
		'CATALOG' log file is an output entry
summtext	N	Specifies the identifier of an SCL list to contain the log's error, warning, and informational messages
err	N	returns the number of error messages
warn	N	returns the number of warning messages
msg	N	returns the number of informational messages

Details

The *summtext* list will be filled with all lines in the log beginning 'ERROR:', 'WARNING:' and 'MESSAGE:'. Note that the application programmer may choose to put MESSAGE messages to log for inclusion in this summary.

2.3.2.2 EMAIL Class

The design calls for a number of simple methods. These are used by the two AUTOMAIL methods and are summarised below:

ADD_ATTACHMENT

Attaches a file to the email message

CALL SEND (*object-id*, 'ADD_ATTACHMENT', *name*);

Where...	Is type...	And...
name	C	specifies the name of the external file to be attached

SEND

Sends the email message

CALL SEND (*object-id*, 'SEND');

Details

The SEND method automatically terminates the object after the message is sent.

WRITE_ENVELOPE

Sets the subject and recipients

CALL SEND (*object-id*, 'WRITE_ENVELOPE', *to*, *subject* <, *cc* >);

Where...	Is type...	And...
to	N	specifies the identifier of an SCL list containing email IDs of recipients
subject	C	specifies the subject of the email message
cc	N	specifies the identifier of an SCL list containing email IDs of users to receive copies of the message

WRITE_MESSAGE_LINE

Writes a line of text to the body of the message

CALL SEND (*object-id*, 'WRITE_MESSAGE_LINE', *line*);

Where...	Is type...	And...
line	C	specifies the text to be written

Details

The body of the email message is built-up using WRITE_MESSAGE_LINE and WRITE_MESSAGE_LINES methods. Lines are added in the order in which they are supplied.

WRITE_MESSAGE_LINES

Writes one or more lines of text to the body of the message

CALL SEND (*object-id*, 'WRITE_MESSAGE_LINES', *lines*);

Where...	Is type...	And...
lines	N	specifies the identifier of an SCL list containing lines of text

Details

The body of the email message is built-up using WRITE_MESSAGE_LINE and WRITE_MESSAGE_LINES methods. Lines are added in the order in which they are supplied.

2.3.3 Maintaining the Control Table

The control table must record a list of recipients for combinations of application, task, and completion code. In the simplest case this can be a SAS data set with four variables (storing the list of recipients in one character variable) though this has obvious limitations.

The list of email recipients must contain identifiers of email users that can be recognised by the email software. Since these identifier may not be simple names it is desirable to interface with the mail system's address book when updating the control table. This is an area that this author has not explored in detail.

2.4 Coding Highlights

This paper is not an appropriate medium for a tutorial in the use of email via SAS software. However, a few notes are appropriate.

2.4.1 Codes from SCL

I had trouble in finding how I could write an email file from SCL. It was eventually pointed-out to me that the codes shown in the table shown below can be written as the first characters in a record. The file to which the records are written must have been opened using the 'EMAIL' device. The message is sent when the file is closed.

!EM_ATTACH!	Attached files
!EM_SUBJECT!	Subject of the email
!EM_TO!	Direct recipients of the email
!EM_CC!	Copied recipients of the email

Multiple attachments and recipients can be specified by enclosing them in parentheses.

The syntax of the email recipients must precisely match that specified by your email client.

2.4.2 Automatic Fileref

When allocating libraries and files, I frequently allow SAS to choose the libref or fileref. You can do this by specifying the LIBNAME() or FILENAME() function's first parameter as a blank character variable. I found a problem with the EMAIL device.

Take a look at the code following this paragraph. You'll see that, for the EMAIL device, SAS/AF generates an invalid fileref. It turns-out that this fileref is usable until you de-assign it (with another FILENAME() function). At that point, the FILENAME() function bombs and tells you that you're using an invalid fileref!

```
length message_fref $8;
init:
  rc=filename(message_fref,'c:\temp\temp.txt');
  put rc= message_fref=;
  message_fref='';
  rc=filename(message_fref,'c:\temp\temp.txt','email');
  put rc= message_fref=;
return;

RC=0 MESSAGE_FREF=_LN00070
RC=0 MESSAGE_FREF=!0000002
```

2.4.3 Large LRECL

All recipients must be coded in big, long lists in two single TO and CC records. To achieve this successfully, the file must be opened with a suitably long logical record length (LRECL).

2.5 In Use - Who's Sending This Email?

When SAS software sends an email, the application tells SAS who to send the mail to but doesn't directly tell SAS who is sending the message. SAS software determines the sender from one of two sources: the EMAIL system options or the active email client.

The EMAIL system options can be specified in the CONFIG.SAS file or when invoking your SAS session. The two key options are EMAILID (specifies the sender's email ID) and EMAILPW (specifies the sender's email ID password). Other options include EMAILSYS, EMAILDLG, and EMAILSERVER. Refer to your system documentation for details.

If EMAILID has not been supplied, the SAS software will attempt to establish communication with a local email client. In the case of a PC running Microsoft Windows, for example, the client could be Microsoft Mail or Lotus cc:Mail. If no client is actively running, the SAS software will attempt to start it.

In order to issue emails at the end of running scheduled tasks, the SAS session running the scheduled tasks must have appropriate EMAIL system options or a mail client must be actively running when the tasks complete. In the first case (setting EMAIL system options), the password of the ID must be hardcoded in the CONFIG.SAS or the command line. Clearly this represents a security weakness. The latter case (active email client) may not be possible if the tasks are not running on a dedicated server or workstation.

3 Non-Volatile Locks

The basic SAS System prevents two users from editing the same data set at the same time. With the addition of SAS/SHARE, two users will be able to edit the same data set at the same time, but not the same record. These two levels of protection fulfil most sites' needs in maintaining data integrity. However, these features are limited to protecting data sets whilst they are actively being edited, i.e. whilst they are open. A number of applications demand the ability to lock a data entity for a wider period of time, including intervals when the data is not being actively edited.

Applications that require this ability include any whereby a user might call across the room to their colleagues "don't update X for the next couple of days, I'm working on it." Applications permitting the construction of technical models or financial budgets are just two such classes. In these cases, SAS software cannot provide the level of protection that is required.

The ability to control concurrent access to any kind of resource is desirable. In this context, resources can be data sets, catalogs, libraries, logical collections of data sets, processes, and application screens. The word "resource" can be used in its widest sense to mean anything that is of use or benefit to the application. Given that definition, the application may have a need to control concurrent access to one or more resources.

With respect to the management of concurrent access, it is not always desirable to simply prevent more than one concurrent access. For example, it may be preferable to allow many users to browse a data entity but only permit one person to edit the entity at the same time. A locking system must provide this flexibility and permit the application to determine the schema to be used for each resource.

3.1 A Brief Analysis

The ability to provide non-volatile locks is not difficult to construct and implement. A fundamental design decision is required with respect to whether the methods actually provide access to the data themselves, or whether they merely act as a set of signals to indicate whether it is safe for the application to access the data. The former provides the greatest degree of safety; the latter provides greater flexibility.

This author has had a great deal of success with implementing a class-based solution using the signalling approach, i.e. not accessing the data but merely indicating to the application that it is safe to do so. In addition to requiring that the class be instantiated (and terminated) by applications using it, the only interfaces are LOCK and UNLOCK methods.

The LOCK method needs a scheme for indicating the resource (data entity) to be locked and the type of access that will be required to it. The method should return an identifier that can subsequently be used for removing the lock.

Given that the resource may be temporarily in use by another user when the LOCK method is issued, the method should communicate with the user to avoid immediate failure of the LOCK attempt. By interacting

with the user, the user can be given the option of waiting and then retrying, of looking to see who currently holds the resource, or of cancelling the request. This interactive failure-management should be optional.

3.1.1 Resource Naming

The scheme used for resource naming should be as flexible as possible. Since the methods will not access the target data themselves, it is possible to derive a naming scheme that will permit control of non-existent resources in addition to real physical resources. That is, the methods can protect data sets before they are created, and can protect logical entities (such as combinations of data sets).

It is desirable to be able to lock all instances of a type of resource, for example an application may wish to lock an individual control table or it may need to lock all financial control tables at the same time. To facilitate this ability, the resource to be locked should be specified as a two-level name, permitting the first level to be a categorical description (such as ‘financial control table’) and the second level to be the name of an individual resource.

3.1.2 Access Levels

The ability to use different locking schemas for different resources is paramount. For some data entities, one user editing the data will mean that nobody else should be able to access it at all; for other types of data entity it will be permissible for users to browse the data whilst one other user is editing it. There are many other permutations that must be permitted.

The locking schemas should not be restricted to using create, read, update, and delete as access types. Different resources will require very different access modes and access rules.

3.2 A Brief Design

3.2.1 Access Levels

To provide the desired flexibility in the access schema, the solution uses combinations of two sharing options. The first sharing option specifies the level of access required by the caller; the second specifies the level of access the caller will permit other users concurrently. The sharing options are specified numerically, and higher numbers represent more restrictive use of the resource.

The numbers used are the choice of the application programmers. This approach performs successfully as long as the numbering scheme is used consistently for any given resource.

Access is granted or declined based upon two simple tests:

1. If the caller’s first sharing option is greater than the second sharing option of any existing user the request is declined
2. If the caller’s second sharing option is less than the first sharing option of any existing user the request is declined

If the request is not declined by either of the preceding two tests the request is granted.

An example is appropriate at this point! Consider a situation whereby a particular data resource can be browsed, edited, or deleted. These three activities can be represented numerically as 1, 2, and 3 respectively.

- If an application wishes to browse the data, and is willing to allow others to edit the data at the same time, the first and second sharing options would be 1 and 2 respectively.
- If an application wishes to edit the same data resource, and is willing to allow others to browse the data at the same time but not concurrently edit, the first and second sharing options would be 2 and 1 respectively.

At run-time, the locking system would resolve the following chronological access requests in the manner described:

User	ShrOpt#1	ShrOpt#2	Result	Comments
A	1	2	Granted	User A permitted to browse the data
B	1	2	Granted	User B permitted to browse the data
C	2	1	Granted	User C permitted to edit the data
D	1	2	Granted	User D permitted to browse the data
E	2	1	Declined	User E cannot edit concurrently with user C

3.2.2 Managing Conflicts

When an application's attempt to lock a resource is declined, the locking system will (by default) present the user with a dialog box informing them of the inability to access the named resource. Thus, by default, control is not returned to the calling application immediately. The dialog box offers three options: Cancel, Retry, and List Holder(s).

The Cancel option returns control to the calling application and informs the application that the request was declined.

The Retry option, as the name suggests, will try to access the resource again. If the attempt is successful, control is returned to the calling application and the application is informed that the request was successful. If the attempt fails, the dialog box reappears with the same options.

The List Holder(s) option displays a list of the users who currently hold a lock on the resource. This may help the user to resolve the conflict by manual means before trying the Retry button.

3.2.3 Method Interface

The design calls for two methods. These are summarised in this section.

LOCK

Applies a lock

CALL SEND (*object-id*, 'LOCK', major, minor, shropt1, shropt2, lockid <,interact >);

Where...	Is type...	And...
major	C	specifies the major name of the resource
minor	C	specifies the minor name of the resource. The keyword '_ALL_' may be used to attempt to lock all of a major resource's minor resources
shropt1	N	specifies the type of access required by the caller
shropt2	N	specifies the type of concurrent access the caller will permit

lockid	N	returns a numeric identifier for the lock: 0 access not granted >0 identifier of the lock
interact	C	specifies whether the user should be interactively queried if the resource is not immediately available: Y query the user (the default) N do not query the user

UNLOCK

Removes a lock

CALL SEND (*object-id*, 'UNLOCK', lockid);

Where...	Is type...	And...
lockid	N	specifies the numeric identifier returned by the LOCK method when the lock was established

3.2.4 Central Control Table

The design hinges upon a central control table that records all active locks. When an application issues a LOCK method the class refers to the control table to establish whether the requested resource is already in use and whether the caller should be granted access. If access is granted then another record is added to the control table. When an application issues an UNLOCK method the requisite record is deleted from the control table.

Clearly, in active systems, the control table may be highly volatile and subject to a high degree of contention. To function correctly it must be opened with member-level access. Thus, in busy systems, the control table should be placed on fast storage media and caching should be considered.

The control table stores the following variables:

- MAJOR
- MINOR
- SHROPT1
- SHROPT2
- USERID
- DATETIME (used as the key)

3.3 Coding Highlights

3.3.1 Opening the Control Table

The highlights of the LOCK method are shown below. In the code shown below, ten attempts to open the control table will be made before the lock request is automatically refused. Careful recoding will give the user the option to retry again instead. For the sake of clarity, some detail has been omitted.

```
granted = 0;
abandoned = 0;
```

```
do until (granted or abandoned);
  open_count = 0;
  do until (abandoned or tableD gt 0);
    open_count = open_count + 1;
    tableD = open('utils.lock (cntllev=mem)','U');

    if tableD gt 0 then do;

      Check to see if lock is available to this caller.
      If so, set granted to a unique value,
      If not then retry. If ultimately not available the set abandoned to 1.

    end;

  end; /* end loop to open data set */

  if open_count gt 10 then abandoned = 1;

end; /* end loop on granted/abandoned */
```

3.3.2 Establishing Whether the Request is Granted

The code shown below is placed within the conditional block described in the preceding section. For the sake of clarity, some detail has been omitted.

```
      /* select all locks on same major and minor resource */
      where = 'upcase(major)=' !! quote(major)
              !! ' and upcase(minor)=' !! quote(minor);

      /* select all locks having ShrOpt1 > user's ShrOpt2              */
      /* (i.e. those locks which will have access beyond that allowed) */
      /* and also those having ShrOpt2 < user's ShrOpt1                */
      /* (i.e. those locks which prohibit the user from having such a  */
      /* high level of access.                                         */
      where = where !! 'also shropt1 gt' !! put(so2,3.) !!
                       ' or shropt2 lt' !! put(so1,3.);

      /* If any locks are found then access cannot be granted.         */

      if where(tableD,where) gt 0 then %fatal;

      if fetch(tableD) eq -1 then do; /* end of data set */
        /*--------------------*/
        /* permission granted */
        /*--------------------*/
        granted = datetime();

        call set(tableD);
        ShrOpt1 = so1;
        ShrOpt2 = so2;
        Userid = own_userid; /* own_userid=instance variable set in _INIT_ */
        datetime = granted;
        if append(tableD) gt 0 then %fatal;
        if close(tableD) gt 0 then %fatal;

      end; /* end granted */

      else do;
        /*-----------------------------------------------*/
        /* permission not granted so interact with user */
        /* if action is cancel then abandon attempt,    */
        /* else recommence looping...                   */
        /*-----------------------------------------------*/
        call display('lk_retry.frame',major,minor,action);
        if action eq 'cancel' then abandoned = 1;

      end; /* end not granted */
```

3.4 In Use - Permanent Locks

It probably won't have escaped your notice that the non-volatile locks can be *very* non-volatile - indeed, permanent. If an application falls over (or is badly written) and does not release the locks that it holds, those

locks will never be released. Thus, the control table becomes full of orphan locks, and the system slowly grinds to a halt as less and less resources remain available.

There's no fully satisfactory solution to this problem. I've tried several routes in the past but I've always had to come back to simply providing an administrator utility to manually remove locks. Typically, this provides the option of deleting all locks held by any one user specified by the administrator. This is possible because the users' userIDs are written to the lock table when locks are applied.[1]

Alternative solutions invariably have a practical reason why they're not feasible. For instance, automatically deleting all of a user's locks when they log-in to the system demands that they be prevented from logging-in more than once at the same time. The best way to prevent users logging-in more than once is to… use the lock system to lock a resource tied to the user when they log-in. The two requirements are mutually exclusive.

4 Conclusion

This paper has described two sub-systems that this author has successfully used for several clients. The sub-systems are generic and can be applied to any application that needs similar facilities.

The need to design and build SAS software applications in a modular fashion is crucial to making effective use of manpower and other resources. This paper emphasises the need for some of those modules to be "back-office" units that form tools for the application developer but are barely surfaced to the end-user. Whilst this is readily apparent on large projects, small projects benefit too. Small projects are simply large projects that haven't yet matured!

The RAD (Rapid Application Development) approach and methodology dominates the SAS software application development industry. Sub-systems like those described in this paper can be more difficult to justify in such an environment because they do not have a direct influence on the time boxes' deliverables. Effort made in the justification, however, is amply repaid in the subsequent application development stages.

ANDREW RATCLIFFE is an international freelance SAS Consultant specialising in SAS/AF application development, Data Warehousing, and the Pharmaceutical industry. He can be contacted by telephone or fax on +44 1322 525672, by email at andrew@ratcliffe.demon.co.uk, and by post at 5 Willow Close, Bexley, Kent, England, GB DA5 1QY.

[1] This would not be possible if users did not need to log-in to the application and their userIDs were not available from the operating system. In practice, however, that situation has never arisen for me.