```
Subj:  Pros and Cons of a separate jmJobStateTable
From:  Tom Hastings, Harry Lewis, and Ron Bergman
Date:  5/14/97
File:  sepstate.doc
```

The biggest issue remaining in the Job Monitoring MIB is the duplication of information in the **jmJobStateTable** and the **jmAttributeTable**. Should we get rid of the duplication? And if so, do we delete the **jmJobStateTable** or the duplicated attributes in the **jmAttributeTable**? A second issue is whether the AssociatedValue object/attribute that provides a discriminant union of values based on the job's state should be kept in either table. This paper is intended to further the discussion about this topic.

1. Summary of current overlap of jmJobStateTable and jmAttributeTable

The overlap between the **jmJobStateTable** and **jmAttributeTable** in the current MIB specification is summarized by the following table:

Table 1

jmJobStateTable object	corresponding **jmAttributeTable** attribute	Mandatory?	static/dynamic?	AssociatedValue state
jmJobState	jobState(3)	yes	dynamic	-
jmJobStateKOctetsCompleted	jobKOctetsCompleted(50)	yes	dynamic	-
jmJobStateImpressionsCompleted	impressionsCompleted(55)	yes	dynamic	canceled
jmJobStateAssociatedValue	jobStateAssociatedValue(4)	yes	dynamic	-
	jobStartedBeingHeldTimeStamp(73)	no	dynamic	held
	numberOfInterveningJobs(9)	yes	dynamic	pending
	jobKOctetsRequested(48)	yes	static	processing
	impressionsRequested(54) [currentCopy(??) proposed]	yes yes	static dynamic	printing printing
	deviceAlertCode(10)	yes	dynamic	needsAttention
	outputBinIndex(34)	yes	dynamic	completed

The **jmJobStateTable** is indexed by **jmJobSetIndex** and **jmJobIndex**.

The **jmAttributeTable** is indexed by **jmJobSetIndex**, **jmJobIndex**, **jmAttributeTypeIndex**, and **jmAttributeInstanceIndex**.

2. Summary of the Issues

The issues around the above objects/attributes are (in a logical order for consideration):

ISSUE 68 **-** Delete the **Job State Group/Table** all together, since all objects are also duplicated as attributes in the **jmAttributeTable**?

Sub-issues to Issue 68 are:

ISSUE 68a: If we keep the **jmJobStateTable**, should we delete the attributes out of the **jmAttributeTable** that already appear as objects in the **jmJobStateTable**, namely **jmJobState(3)**, **jobKOctetsCompleted(50)**, and **impressionsCompleted(55)**?

ISSUE 68b: If we keep the **jmJobStateTable**, should we move the *mandatory* associated attributes (1) out of the **jmAttributeTable** that the **jmJobStateAssociatedValue** object provides a convenient copy and (2) into the **jmJobStateTable** as objects? Then the **jmAttributeTable** would contain only conditionally mandatory attributes and the **jmAttributeTable**, itself, would change from Mandatory to Conditionally Mandatory.

In other words, move **numberOfInterveningJobs**(9), **jobKOctetsRequested**(48), **impressionsRequested**(54), [or the proposed **currentCopy**(??)], **deviceAlertCode**(10), and **outputBinIndex**(34) into the **jmJobStateTable** as *mandatory* objects: **jmJobStateNumberOfInterveningJobs**, **jmJobStateKOctetsRequested**, **jmJobStateImpressionsRequested** [or proposed **jmJobStateCurrentCopy**], **jmJobStateDeviceAlertCode**, and **jmJobStateOutputBinIndex**. (Don't move the non-mandatory **jobStartedBeingHeldTimeStamp**(73)).

ISSUE 69**-** Does order of assignment of **JmAttributeTypeTC** enums make any difference?

Would it help if the mandatory attributes were first, so that Get Next would pick them up first when getting the next conceptual row? Does making the attribute table easier to navigate using Get Next help with the decision to Issue 68 and 68b?

ISSUE 75 **-** Should the Attribute enum values be grouped so additions could be added in the appropriate section

When producing the first Internet-Draft, I re-arranged the Attribute enums into logical groups, so that attributes would be easier to find. We now have 78 attributes, so logical grouping is becoming important to make the list more understandable. Several people had proposed adding attributes that were already present in the spec. Also Harry has expressed the concern that any re-assignment of at least OIDs, causes problems with tracking the drafts Finally, when the standard achieves proposed status, there will be additional registrations. It might be helpful if the enums could be assigned to the appropriate group, instead of only at the end.

The current logical grouping are:

Job State attributes	10
Job Identification attributes	19
Job Parameter attributes	7
Image Quality attributes (requested and used)	6
Job Progress attributes (requested and consumed)	7
Impression attributes (requested and consumed)	6
Page attributes (requested and consumed)	3
Sheet attributes (requested and consumed)	3
Resource attributes (requested and consumed)	7
Time attributes (set by server or device)	9

OK to assign Job State and Job Identification in steps of 30 and the rest in steps of 20?

See also Issue 69. We could put the mandatory attributes first, and then group the rest as above.

Issue 78 **-** Should the "multiplexor" (discriminant union?) **jobStateAssociatedValue(4)** attribute be removed from the Job Attribute Table and the equivalent **jmJobStateAssociatedValue** object be removed from the Job State table?

The associated values are also available as attributes in the attribute table. The application has to either (1) request all 7 associated attributes or (2) first request the **jobState**(3) attribute and the request the 1 pertinent attribute. Since all 7 will easily fit in a PDU (minimum of 500 octets or so on all systems) and each request takes about 20 octets, so you can get about 20 (5*4) attributes into a single PDU.

Issue 79 **-** Should the **'printing'** state be combined into the **'processing'** state?

Many printers don't distinguish between '**processing'** and **'printing'**, especially desktop printers. For those that do, having a state change that really reflects progress, such as the transition from processing to printing, is better handled as a job state reason, not as a fundamental state change. Finally, since this MIB is intended for non-printing services in the future, such as fax out, CD-ROM writing, fax-in, scanning, etc., it would help if one of the states wasn't **'printing'**. Even IPP, only has the state of **'processing'**, with a job-state-reason of '**job-printing**' for those implementations that make the distinction and want to go to the trouble of indicating the difference. IPP even indicates that "most implementations won't bother with this nuance".

ISSUE 68c: If we keep the **jmJobStateAssociatedValue** object, we could just change the attributes listed in ISSUE 68b from *mandatory* to *optional* and keep them only in the **jmAttributeTable**. The **jmJobStateAssociatedValue** object would remain in the **jmJobStateTable** to provide access to these attributes mandatorally.

Issue 76 **-** So should **jobName**, **jobOwner**, and one of **deviceNameRequested** or **queueNameRequested** be made Mandatory?

When we moved attributes from the job table to the attributes table (Issue 54 and 56), we didn't make any of them mandatory for an agent to implement. Should any of them be made Mandatory?

The old job table had the following (mandatory) objects in it:

jmJobName
jmJobIdName
jmJobIdNumber
jmJobServiceType
jmJobOwner
jmJobDeviceNameOrQueueRequested
jmJobCurrentState
jmJobStateReasons

1. **jmJobIdName** and **jmJobIdNumber** have been replaced by **jmJobSubmissionIDIndex** which is Mandatory.
2. **jmJobServiceType** need not be Mandatory.
3. Also **jmJobDeviceNameOrQueueRequested** has been made into two separate attributes: **deviceNameRequested** and **queueNameRequested**, so we'd have to make either one of them mandatory.
4. **jmJobCurrentState** is now **jobState** and is Mandatory
5. **jmJobStateReasons** became four attributes: **jobStateReasons1, jobStateReasons2, jobStateReasons3,** and **jobStateReasons4.** None of them need to be Mandatory.

So should **jobName**, **jobOwner**, and one of **deviceNameRequested** or **queueNameRequested** be made Mandatory?

ISSUE 76a **-** If yes, then should they be put into the **jmJobStateTable**, instead of the **jmAttribute** table, if Issue 68b concluded that the **jmAttributeTable** should have no mandatory attributes.

ISSUE 70 **-** Add some simple general device alert TC, instead of using the Printer MIB **Alert** Codes.

The **PrtAlertCodeTC** generic values are *not* much good to an end user without knowing which subunit. For example, **SubUnitEmpty** isn't very informative by itself. If an implementation also has the Printer MIB, then a lot more information is available, so a copy of the Printer Alert isn't very useful. If the implementation doesn't have the Printer MIB, then the Printer Alert codes aren't informative enough.

Even worse, the deviceAlertCode(10) is Mandatory, which can't be implemented, if there isn't a Printer MIB also implemented.

Issue 73 **-** Is there a problem with **outputBinIndex** being made mandatory?

If **outputBinIndex** is made mandatory, but an implementation doesn't have the Printer MIB, the agent has to put 0 as the value. Should we add one more attribute: **outputBinNumber**, which is just a number, not an index into the Printer MIB? If we do, which should be mandatory? Just one more reason to get rid of the **jmJobStateTable**, which is forcing us to pick a particular outputBin implementation and make it mandatory. If we got rid of the **jmJobStateTable**, we could forget about making any of the 3 outputBinName, outputBinNumber, or outputBinIndex attribute mandatory.

Closed: Don't add **outputBinNumber**. Just add **other**(-1), **unknown**(-2), and **multi**(-3) values and keep **outputBinIndex** as mandatory. This does also means that **jmAttributeValueAsInteger** needs a lower bound of -3, not -2.

ISSUE 87 **-** When shall an agent make the mandatory attributes appear in the **jmAttributeTable**?

Shall an agent materialize all mandatory attributes when the job is submitted, so that a requester can access them all with multiple explicit Gets in a single PDU, without fear of a missing object aborting the PDU? If the mandatory attributes are represented as objects in the jmJobStateTable, then it is clear from SNMP rules that the agent shall materialize at least an empty value for each mandatory object (attribute).

ISSUE 83 **-** Can some attributes be deleted before the **jmGeneralAttributePersistence** expires?

Harry Lewis' 5/2 e-mail suggested that some of the attributes, such as "**numberOfInterveningJobs**(9)" don't even need to persist the shorter time specified by **jmGeneralAttributePersistence**.

However, if we move the mandatory attributes to the **jmJobStateTable** and make them objects, then they shall persist for the longer persistence specified by **jmGeneralJobPersistence**.

See the rest of the issues list for the issues that do *not* relate to the overlap objects/attributes between the **jmJobStateTable** and the **jmAttributeTable**.

3. Accessing the jmJobStateTable and the jmAttributeTable

In order to understand the pros and cons, it seems necessary to understand how an application would use Get and Get Next to get information from these two tables. We need to consider the three basic types of applications: (1) a job monitoring application that is monitoring a particular job, (2) a job monitoring application that is monitoring all jobs on a device or server, and (3) a job accounting or utilization program. The first two kinds of applications are interested in active jobs and the third is interested in inactive jobs (canceled, or completed).

3.1 OID assignments to the objects

In order to construct complete examples, it is helpful to use the actual OIDs that will be assigned to the objects and attributes in the MIB:

```

>         jobmonMIB
>         jobmonMIB.1 jobmonMIBObjects
>         jobmonMIB.1.1 jmGeneral
          jobmonMIB.1.1.1 jmGeneralTable
          jobmonMIB.1.1.1.1 jmGeneralEntry
          jobmonMIB.1.1.1.1.1 jmGeneralNumberOfActiveJobs
          jobmonMIB.1.1.1.1.2 jmGeneralOldestActiveJobIndex
          jobmonMIB.1.1.1.1.3 jmGeneralNewestActiveJobIndex
          jobmonMIB.1.1.1.1.4 jmGeneralJobPersistence
          jobmonMIB.1.1.1.1.5 jmGeneralAttributePersistence
          jobmonMIB.1.1.1.1.6 jmGeneralJobSetName

>         jobmonMIB.1.2 jmJobID
          jobmonMIB.1.1.1 jmJobIDTable
          jobmonMIB.1.1.1.1 jmJobIDEntry
          jobmonMIB.1.1.1.1.1 jmJobSubmissionIDIndex
          jobmonMIB.1.1.1.1.2 jmJobSetIndex
          jobmonMIB.1.1.1.1.3 jmJobIndex

>         jobmonMIB.1.3 jmJobStateG
          jobmonMIB.1.1.1 jmJobStateTable
          jobmonMIB.1.1.1.1 jmJobStateEntry
          jobmonMIB.1.1.1.1.1 jmJobState
          jobmonMIB.1.1.1.1.2 jmJobStateKOctetsCompleted
          jobmonMIB.1.1.1.1.3 jmJobStateImpressionsCompleted
          jobmonMIB.1.1.1.1.4 jmJobStateAssociatedValue

>         jobmonMIB.1.4 jmAttribute
          jobmonMIB.1.1.1 jmAttributeTable
          jobmonMIB.1.1.1.1 jmAttributeEntry
          jobmonMIB.1.1.1.1.1 jmAttributeTypeIndex
          jobmonMIB.1.1.1.1.2 jmAttributeInstanceIndex
          jobmonMIB.1.1.1.1.3 jmAttributeValueAsInteger
          jobmonMIB.1.1.1.1.4 jmAttributeValueAsOctets

>         jobmonMIB.2    jobmonMIBConformance
          jobmonMIB.2.1  jobmonMIBCompliance
          jobmonMIB.2.2  jmMIBGroups
          jobmonMIB.2.2.1 jmGeneralGroup
          jobmonMIB.2.2.2 jmJobIDGroup
          jobmonMIB.2.2.3 jmJobStateGroup
          jobmonMIB.2.2.4 jmAttributeGroup

```

3.2 Tables and the Get operation

Recall that the OIDs for table entries consist of the OID for the entry (column) in the table, *followed* by the index(es) to that entry. To get the job state object in the **jmJobStateTable** for the job with a **jmJobIndex** of 1000 in job set 1, the requester must pass the following OID as a Get input parameter:

jmJobState.1.1000, i.e., jobmonMIB.1.1.1.1.1.1.1.1000.

To get the corresponding from the **jmAttributeTable,** which is the **jobState**(3) attribute, the requester must pass the following OID as a Get input parameter:

jmAttributeValueAsInteger.1.1000.3.1, i.e., jobmonMIB.1.1.1.1.2.1.1000.3.1.

Thus an application can always get the corresponding attribute from the **jmAttributeTable** with an OID that is only *two* octets longer than is required on a Get for the corresponding object **jmJobStateTable**.

An application can get multiple objects from the **jmJobStateTable** and can get multiple attributes from the **jmAttributeTable** by supplying multiple Get operations in a single PDU.

If there is no such object, the Get operation returns an error (and does *not* perform any further Get operations in the submitted PDU, correct?)

3.3 Tables and the GetNext operation

The SNMP GetNext operation returns the value of the object specified by the *next lexically higher* OID from the one supplied as an input parameter. GetNext also returns that next lexically higher OID itself, so that the application can pass it back as an input parameter to a subsequent GetNext in order to get the next object. If there are no lexically higher objects, GetNext returns an error.

The OID input parameter does not need to be "fully specified". Trailing OID arcs can be omitted and they shall behave as if the requester supplied 0 for those arcs.

For a single index table, Get Next can be used to get the "next conceptual row" in the table. GetNext must be used when the agent scatters rows in a table, i.e., the table is a "sparse" table. MIB specifications can specify that tables shall not be sparse. The **jmJobStateTable** is specified such that agents shall enter conceptual rows such that jmJobIndex is monatonically increasing, until wrap occurs. However, because jobs may be canceled, a canceled job may be removed from the middle of the table (after persisting for the **jmGeneralJobPersistence** time), thereby making the **jmJobStateTable** have an empty row, i.e., be a "little bit sparse". Also a system that processes jobs out of order may result in some empty rows in between rows that are awaiting the **jmGeneralJobPersistence** time to expire.

An application can get the state of the next job after job 1000 in the **jmJobStateTable** by passing in the (same) OID:

jmJobState.1.1000, i.e., jobmonMIB.1.1.1.1.1.1.1.1000

If job 1001 had been canceled, say, and the agent removed it subsequently, the agent might return the state of job 1002 and the OID:

jmJobState.1.1002, i.e., jobmonMIB.1.1.1.1.1.1.1.1002

The application could copy the returned OID to the input parameter of a subsequent GetNext and the get the state of the next job after 1002, and so forth.

If the application wanted to get more than just one object in the next conceptual row, the application could supply several GetNext operations in the same PDU. So to get the **jmJobState**, **jmJobStateKOctetsCompleted**, **jmJobStateImpressionsCompleted**, and **jmJobStateAssociatedValue** objects from the **jmJobStateTable** for the next job after 1002, the application could pass in the following four OIDs in four Get Next operations in the same PDU:

jmJobState.1.1002, i.e., jobmonMIB.1.1.1.1.1.1.1.1002
jmJobState**KOctetsCompleted**.1.1002, i.e., jobmonMIB.1.1.1.1.2.1.1002
jmJobState**ImpressionsCompleted**.1.1002, i.e., jobmonMIB.1.1.1.1.3.1.1002
jmJobState**AssociatedValue**.1.1002, i.e., jobmonMIB.1.1.1.1.4.1.1002

The agent shall *return* the next OID in each GetNextResponse for each of these inputs, which would be the corresponding column in the next row, say, job 1003:

jmJobState.1.1003, i.e., jobmonMIB.1.1.1.1.1.1.1003
jmJobState**KOctetsCompleted**.1.1003, i.e., jobmonMIB.1.1.1.1.2.1.1003
jmJobState**ImpressionsCompleted**.1.1003, i.e., jobmonMIB.1.1.1.1.3.1.1003
jmJobState**AssociatedValue**.1.1003, i.e., jobmonMIB.1.1.1.1.4.1.1003

NOTE **-** An application could *not* perform the above by using a individual repeated GetNext operation copying each result to the single input parameter, because GetNext increments the least significant part of the OID first. Thus, each individual GetNext would get the *same* column in the next row, *not* step across the columns in the same row.

In order to perform the equivalent of the above example in the **jmAttributeTable**, i.e., get the **jobState**(3), **jobKOctetsCompleted**(50), **impressionsCompleted**(55), and the **jobStateAssociatedValue**(4) attributes in the **jmAttributeTable** for the next job after job with **jmJobIndex** 1002, the application must first determine the next valid jmJobIndex, which cannot be done by simply passing in the following OID in GetNext operation:

jmAttributeValueAsInteger.1.1002.3.1, i.e., jobmonMIB.1.1.1.1.2.1.1002.3.1

because the next lexically higher OID might be:

jmAttributeValueAsInteger.1.1002.9.1, i.e., jobmonMIB.1.1.1.1.2.1.1002.9.1

which is the **numberOfInterveningJobs**(9) attribute.

Instead, the application must determine what the next **jmJobIndex** value either by doing a GetNext on the **jmJobStateTable** or by passing in the "incremented" partial OID that the application has incremented "by hand" and shortened by removing the trailing OID arcs after the **jmJobIndex** arc:

jmAttributeValueAsInteger.1.1003, i.e., jobmonMIB.1.1.1.1.2.1.1003

which will return the first attribute in the next job. The **jmJobIndex** arc the comes back in that GetNextResponse is the next **jmJobIndex** in the **jmJobAttributeTable**.

3.4 Monitoring a single specific job

When a user submits a job, the client could fire up a monitoring application that monitors the job just submitted. The monitoring application needs to determine the job's **jmJobIndex** by one of several methods, depending on the implementation and the configuration:

(1) is told the **jmJobIndex** of the job to be monitored because the server returned the job-identifier which the application knows the map to **jmJobIndex** value,

(2) can determine the **jmJobIndex** by doing a Get supplying the OID for the **jmJobSubmissionIDIndex** to the **jmJobIDTable** as follows. Suppose that the job submission id generated by the client is: "12345678nnnnnnnnnn"

jmJobIndex.1."12345678nnnnnnnnnn", i.e., jobmonMIB.1.1.1.1.3.1."12345678nnnnnnnnnn"

which returns the **jmJobIndex** for the job, or

(3) can scan the **jmAttributeTable** looking for attributes that match, such as **jobOwner**(15), **jobName**(13), etc., though such a scan requires two probes: first to find the next **jmJobIndex** either from the **jmAttributeTable** or more straightforwardly from the **jmJobStateTable**.

Give the jmJobIndex for the single job being monitored, the application can use direct Get operations to get any objects from the **jmJobStateTable** or attributes from the **jmAttributeTable** as shown above.

3.5 Monitoring all active jobs on a server or device

An operator might run an application that monitors all *active* jobs on a server or device. Such an application polls at some frequent enough interval to show changes, but not too frequently to bog down the network or server/device. An end-user might fire up an application to monitor all jobs on a server or printer, especially when searching for a "least busy printer". Here the time to find the jobs and get their attributes needs to be relatively short, or the user will not want to fire up such an application.

With either scenario, the application has to determine the oldest active job with a Get specifying the jmJobSet=1, and it may as well get the number of active jobs and the newest active job index in the same PDU:

jmGeneralNumberOfActiveJobs.1, i.e., jobmonMIB.1.1.1.1.1.1
jmGeneralOldestActiveJobIndex.1, i.e., jobmonMIB.1.1.1.1.2.1
jmGeneralNewestActiveJobIndex.1, i.e., jobmonMIB.1.1.1.1.3.1

If the value of **jmGeneralOldestActiveJobIndex** is 0, there are no active jobs and the application updates the display to show no jobs.   Say the value of **jmGeneralOldestActiveJobIndex** is 2000.

Then the application requests, say, the four (column) objects in the **jmJobStateTable** with four Gets in a single PDU as shown above for job 1000. Then the application submits four GetNext operations in the same PDU for each of the four objects in the **jmJobStateTable** as described above for job 1002.

Finally, if there are some additional attributes that the application wishes to get, such as **jobStateReasons1**(5) and **jobName**(13), the application submits several Gets in a single PDU of the form:

jmAttributeValueAsInteger.1.2000.5.1, i.e., jobmonMIB.1.1.1.1.2.1.2000.5.1

jmAttributeValueAsOctets.1.2000.13.1, i.e., jobmonMIB.1.1.1.1.2.1.2000.13.1

3.6 Accounting/Utilization application gathering data on completed/canceled jobs

The accounting or utilization application remembers the lowest jmJobIndex from last time. The application can either get all **jmJobStateTable** objects and all **jmAttributeTable** attributes, or may get only certain selected attributes.

To get all attributes, that application starts with the lowest jmJobIndex that it had on the previous poll cycle and supplies a number of GetNext operations in a single PDU.

To get only selected attributes the application must first determine the next **jmJobIndex** by using GetNext on the **jmJobStateTable**. The application may as well get the other objects from the jmJobState with a bunch of GetNext operations in the same PDU. If the job is active, that data is probably thrown away, and the application steps on to the next job. If the job is inactive (canceled or completed), then the application would specify multiple Get operations in a single PDU, one for each attribute that it wished.

4. Conclusions

The **jmJobStateTable** is very useful because its lowest order index is **jmJobIndex**, so that any number of selected objects can be obtained with multiple GetNext operations in a single PDU for the *next* job, skipping over jobs that have been removed from the table. A subsequent PDU can contain multiple Get operations for any attributes desired using the returned **jmJobIndex** value.

If the mandatory attributes are all put into the **jmJobStateTable** as objects, and not in the **jmAttributeTable** as attributes, it is clear by SNMP rules that all of the mandatory objects shall be instantiated at the same time when the new job row is put into the **jmJobStateTable**. Also the persistence time is clearly separated by which table the information is contained.

The **jmAttributeTable** only contains conditionally mandatory attributes, no mandatory attributes, so that the **jmAttributeTable** itself can be conditionally mandatory, thereby allowing a very small implementation to only implement the **jmJobStateTable** and not the **jmAttributeTable**.