WebSphere Application Server

WebSphere Liberty Batch: Understanding the SMF 120 Subtype 12 Record

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102668** under the category of "White Papers"

Version Date: April 9, 2019
See "Document change history" on page 18 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software

Written by:
David Follis
IBM Poughkeepsie
845-435-5462
follis@us.ibm.com
Don Bagwell
IBM Advanced Technical Sales
301-240-3016
dbagwell@us.ibm.com

Many thanks go to Andy Mauer and Dan Belina for writing the code and all the people who contributed comments to the design.

Table of Contents

Introduction

In Liberty 16.0.0.3 support was introduced for SMF recording for JSR-352 Java Batch jobs. An SMF 120-12 record is written at the end of each step in the job as well as at the end of the job itself. In this paper we'll take a look at the contents of that record along with some hopefully useful comments about where the data comes from and how you might make use of it.

But before we get into that, it might be helpful to take a look at IBM's history with Java Batch in WebSphere and how SMF recording was handled previously.

UPDATE: In Liberty 17.0.0.1 the Liberty Batch SMF 120-12 record was updated to Version 2. Look for text with this dark bar on the left edge to indicate updates for V2. New fields in the tables are marked with “(V2)” because the bar on the edge made things messy.

UPDATE: In Liberty 19.0.0.4 the Liberty Batch SMF 120-12 record was updated to Version 3. This version changed the length of the Subsystem Section because it introduced a new flag word into that section (whose version also changed to 3).

SMF Recording in WebSphere Compute Grid

WebSphere Compute Grid is IBM’s original Java Batch implementation that was delivered as a product separate from the application server (originally included in a product called WebSphere XD (eXtended Deployment) but later broken out on its own). Ultimately it merged into the base WebSphere Application Server product. It runs as part of the traditional application server and not in the newer Liberty server.

Jobs were dispatched in Compute Grid via an HTTP request from the scheduler into a server acting as an endpoint. The servlet that executed used a WorkManager to run an Async Bean which was the batch job. The bean ran on a thread and when it was done, the job was finished.

In the first pass at providing SMF reporting for Compute Grid jobs, the SMF 120 subtype 20 record was assigned. WebSphere owns the start of the subtype range for the 120 record. The 120-20 was written at the end of the job and contained the following information:

```
jobID
jobState
jobNodeName
jobServerName
jobSubmitter
jobAccnting
jobStartTime
jobLastUpdated
zosJobCpuTime
zosJobCPOnlyTime
```

This provides some very basic information. We have an identifier, a state, where the job ran (node/server), who ran it, an accounting string from the job itself, a start time and last updated time (probably the end time), and z/OS CPU information on regular and specialty (zAAP/zIIP) processors.

Meanwhile the application server itself was writing SMF 120-9 records. The subtype 9 contains a great deal of information about regular requests dispatched in the server. In fact a 120-9 would be written for the servlet that was used to launch the batch job. But there was no SMF record that had information about async beans.

That changed in WebSphere Application Server Version 8.0. With V8, WebSphere introduced a slightly different flavor of the 120-9 record that reported on async work. For async work the record included all the server identification information that was in a 'normal' 120-9 record, but none of the request-based information, the classification data, the network data, or the security data. But it did include a new section with information about the async work, including the CPU time. The async flavor also included the "user data" section that was available in 120-9 records written for 'normal' requests.

Information could be placed in the user-data section by an application using an API provided by WebSphere. Remember that at this point Compute Grid was still a separate product from the application server and thus was viewed as an 'application' to the server. It was decided that Compute Grid V8.0 would change from writing its own 120-20 records to just add job-specific user-data to the 120-9 record written for the async work that is the batch job. This gives us a lot more information about the job and where it ran (asid, tcb, TCB CPU time vs. enclave CPU time, etc).

But how would we tell Compute Grid user data from some other user data? Fortunately the user data is in two parts. One part is, of course, the blob of data itself. The other part is an integer tag that can be used to identify WHAT user data it is. IBM reserved a range of those tags for its own use (the reserved range is from 0-65535). Compute Grid chose to use the user data tag of '101'. The reason for that is lost to history.

Since so much information was already provided by the 120-9 itself, the Compute Grid user data only needed to provide information that was specific to the job. Thus the data contains only the job id, the submitter, and an accounting string (if present).

For more information about the contents of the 120-9 record see the Knowledge Center and the whitepaper WP101342.

A Quick Overview of Liberty Batch

JSR-352 specifies that a job may be defined in XML to have one or more steps with conditional flow between the steps. The XML file (called JSL – Job Specification Language) can define a step to be either a simple batchlet step (just run this Java program) or a chunk step. In a chunk step a reader, processor, and writer class are called iteratively with results and checkpoint data persisted via transactional updates.

The job can also define partitioned steps wherein multiple copies of a single step are run concurrently on different threads (possibly in different servers in the IBM implementation) each potentially processing different ranges of input data.

The JSL can also identify a series of steps as a group called a flow. Separate flows can be marked as able to be run concurrently as a split. The flows run on separate threads from the main job thread. You can nest split/flows inside a flow of another split/flow

The IBM implementation provides for a single and multi-JVM configuration. In the single-JVM configuration jobs are run in the server to which they are submitted. In the multi-JVM configuration jobs are submitted to a dispatcher which places the job on a message queue. Messages (and thus jobs) are selected by job properties from the queue to be run in appropriate executor servers.

A good starting point for more information is the WP102544 article on IBM techdocs (http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102544).

What about the Type 30's for Liberty?

Traditional JCL batch jobs on z/OS will rely on the content of the SMF Type 30 records for information about the job, the steps, resource usage, etc. Type 30 records are produced for every address space on z/OS. Does that help with monitoring Liberty Batch jobs?

The Liberty server itself is an address space running on z/OS. Type 30 records will be written for the server just like everything else. However, those records will report on the entire address space. They will include all the CPU information and all the storage and I/O data, but it is all reported for the entire process.

z/OS has no visibility to what is going on inside the address space on a per-thread basis. z/OS has no idea that some of the threads are JVM garbage collection threads and some of the threads are running jobs. It has no idea which jobs are on which threads or how dataset allocation might relate to any particular job.

Remember too that from a memory usage perspective z/OS sees the JVM as having acquired a large chunk of storage early on and it seems to have been using it all. Memory usage within the JVM heap and subsequent garbage collection isn't understood by z/OS itself.

SMF for Liberty Batch

Our original thought was that we would emulate what Compute Grid had done and add user data to the SMF 120-11 record being created for Liberty. But it turns out that the 120-11 records will only cover the dispatch of the REST request used to submit the job and not the job itself. The Liberty Batch job isn't run as an async bean like it was in Compute Grid. The hooks to do SMF recording were going to have to be put directly into the Liberty Batch code.

This meant that we needed to create all new code to write a brand new SMF record. For that reason Liberty Batch jobs are recorded using the SMF 120-12 record.

Because we weren't riding on top of an existing record, we could start from scratch and build the record any way we wanted. Given that customers use SMF Type 30 records for traditional JCL batch jobs, we made an effort to model the 120-12 on the Type 30.

There are a lot of differences in the way a Liberty Batch job runs compared to a JCL job that will make the record content different. Some data available in the Type 30 just isn't available in a multi-threaded Liberty environment where multiple jobs are running concurrently in the same address space. Hopefully the 120-12 will at least feel a little familiar to those used to managing JCL batch jobs with the Type 30s.

Version Date: Tuesday, April 09, 2019

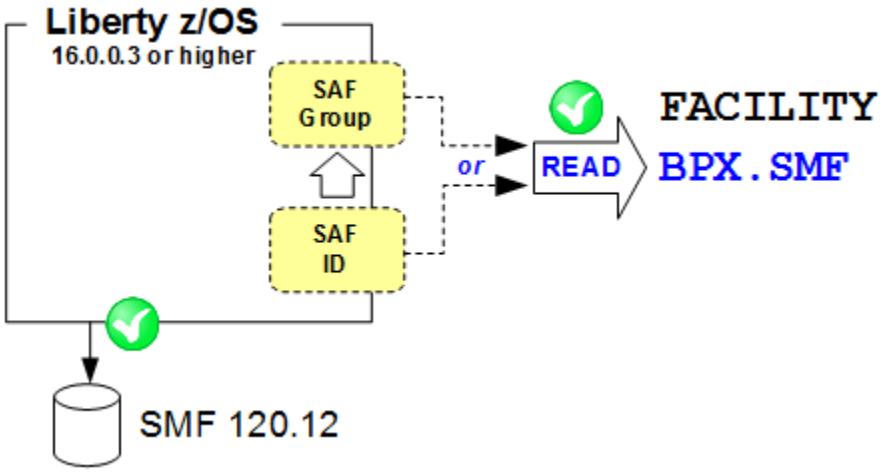
Enabling Liberty SMF Recording

In WebSphere traditional to enable SMF recording you needed to set some configuration options for the server and make sure the `SMFPRMxx` parmlib member you were using allowed SMF 120 records to be recorded (and that you didn't have an SMF exit suppressing them...).

WebSphere Liberty is the same (SMFPRMxx allows SMF 120, and no SMF exit suppressing them) and is also a bit different. Like pretty much everything in Liberty, SMF recording is an optional feature. If you add the `batchSMFLogging-1.0` feature to the featureManager clause of your `server.xml` the server will *try* to write SMF records for Liberty Batch jobs.

But that's not all you need to do. Writing SMF records requires the calling program to be APF authorized. However, a Liberty server is generally not running as an APF authorized program. We could have made use of the Angel to get authorized (as Liberty does for other system functions). But we decided we didn't want to require the Angel to get SMF records. WebSphere traditional didn't have this concern since SMF records were written by the controller which always ran APF authorized.

Fortunately for Liberty, LE/USS provides an `smf_record( )` API that can be called by unauthorized programs to write SMF records. To make this work the identity of the server has to be permitted READ access to the `BPX.SMF` entity in the `FACILITY` class:



Therefore you also need to ensure that whatever userid (or group) the server is running under is granted this access. If the server is a started task then the started task identity is the one you want. If the server is started from the shell using the 'server' command then it will be the userid of the shell.

Formatting Liberty SMF Records

You have a Liberty server at least at level 16.0.0.3 with the `batchSMFLogging-1.0` feature installed. You configured the feature. You granted the right SAF permissions. You ran some Liberty Batch jobs through the server. Parmlib is set up to allow SMF 120-12 records and no obscure SMF exits you no longer have the source for are blocking the records from being written.

You run the SMF dump utility and behold... SMF 120 records. Now what?

We've updated our little Java program (`bbomsmfv.jar`) that formats the WebSphere SMF records to process the new 120-12 records. That's not a bad place to start just to verify you are actually getting records and the contents look right.

> The program was updated to handle the Version 2 records. Be sure to get a new copy if you start writing the new records.

If you want, you can also write Java plugins for the IBM formatter to produce your own reports. Or if you use a vendor product to handle other SMF records, check with your vendor to see if they support the 120-12 records.

Record Contents

For each Type 120 Subtype 12 record, no matter what the batch-record-type is (job-end or step-end), we are going to have almost all the same sections. The mapping of the sections is the same no matter why we are writing the record. Some batch-record-types may not have all the fields filled in. It might be that the meaning of the content of a particular field may vary depending on why we are writing the record (e.g. the ‘start time’ will be the start time for the job or for the step depending on whether we are writing a job-end or a step-end record).

The type 120-12 will have the following sections that are similar to Type 30 sections:

- Subsystem
- Identification
- Completion
- Processor
- Accounting
- USS

We will go through the contents of those one by one coming up.

For completeness, we are not mapping these Type 30 sections into the 120-12:

- I/O Activity
- Storage
- Performance
- Operator
- EXCP

Version Date: Tuesday, April 09, 2019

- APPC resources
- Usage
- ARM
- Multi-system enclave
- Counter

Now we'll go through the contents for each of the sections we do have.

Contents – The Subsystem Section

The first section in the record is always present and there is just one instance of it. This section identifies the Liberty server that wrote the record and provides some other generic information that might be useful in processing the rest of the record. Much of the content of this section matches the server identification section from the Liberty SMF 120-11 record.

Field	*Description*	*Length*[1]
SM120CBD	Version	4
SM120CBE	Record Type (job-end, step-end)	4
SM120CBF	System Name (CVTSNAME)	8
SM120CBG	Sysplex Name (ECVTSPLX)	8
SM120CBH	Timezone Offset (CVTLDTO)	8
SM120CBI	Java Timezone	32
SM120CBJ	Server Job id (JSABJBID)	8
SM120CBK	Server Job name (JSABJBNM)	8
SM120CBL	Server Stoken (ASSBSTKN)	8
SM120CBM	Server ASID (ASCBASID)	4
SM120CBN	Server Config Directory	128
SM120CBO	Server Product Version	16
SM120CDU	RTCPCPUA (V2)	4
SM120CDV	RMCTADJC (V2)	4
SM120CDW	Repository Type (V2)	4
SM120CDX	Job Store Ref ID (V2)	16
SM120CEI	Flag word (V3)	4

After the section version we have the record type. This field will tell you whether the record was written for the end of a step (1) or the end of a job (2). Later fields in the record will help you match the step-end records with the job-end record for the job the steps were part of.

Starting with the Version 2 record the record type can also indicate a partition-end record (3), a flow-end record (4), or a decider-end record (5). The partition end record is important because it is

1 Length is expressed in decimal format.

the only way to capture CPU used by partitions since they don't run on the same thread as the step which spawned them and are thus not included in CPU in the step-end record. The flow-end record won't contain any CPU information because that information is already included in the steps that ran as part of the flow.

A decider is a special step that runs when a split ends and the flows come back together.

Next we have the system and sysplex names where the server is running. The uniqueness of these names depends, of course, on how unique the names are in your environment.

Following that is some information about the timezone where the server is running. We have both the GMT offset value, taken from a z/OS system control block, and the Java string describing the timezone.

Next we include the JES jobname and job id of the server (i.e. `BBGZSRV` and `STC00824`). The job id is kind of unique, but the jobname might be a key to identifying the server. Maybe.

For the geeky identifiers we also include the ASID and STOKEN of the server. ASID values get reused of course, so it isn't a great identifier. But it might help correlate with other records that contain an ASID. The STOKEN is unique within the IPL which is hopefully not something that happens very often. It is less commonly used, but more unique.

Finally, we include the server code level (i.e. 16.0.0.3) and server configuration directory. You will notice we have **not** included the server name. The server name is really just the name of the actual directory where the `server.xml` lives. If my `server.xml` is located at:

```
/u/follis/servers/server1/server.xml
```

then the 'name' of my server is 'server1'. But if my neighbor Tim has a server at:

```
/u/tim/servers/server1/server.xml
```

then the name of his server is *also* 'server1'.

To have the real name of the server you need *both* the directory name AND the path that got you there. That's why we include the whole path to the directory where `server.xml` lives (field `SM120CBN`, "Server Config Directory"). Note that this field is a maximum of 128 characters. If the path is longer than that, we truncate on the left, keeping the likely more interesting bits towards the end of the path.

In Version 2 of the record we also added four new fields to the end of this section. The first two are to help you do appropriate math with the CPU usage information available elsewhere in the record. We have provided the RTCPCPUA and RMCTADJC values. I'm not going to go into what these values mean and how you use them here. There are other resources online to help you with that.

We also added a field to let you know what type of Job Repository is being used by the server that executed this job. There are only two choices, JPA or MEM. The first indicates that we are using Java Persistence API to interact with a real database. The second indicates that we are using an in-memory Job Repository. Why do you care? Well, the various job identifiers that we will see in the Identification Section are only unique within the Job Repository. So if you are using an in-memory repository then those identifiers are only unique within the server running the job. You could very easily have SMF records from multiple servers using in-memory repositories that have identical identifier values.

But what if you are using a real database (indicated by a value of 'JPA')? Well then, the identifiers are unique among servers sharing the same set of tables as a Job Repository. Could you have several different repositories used by different servers? Sure! Well...that's a mess then, isn't it? How can you know what servers are using which tables in which database? To help you out with that we've added the identifier of the datasource used for the Job Repository in the server.xml of

the server into the SMF record. What's that mean? Well, in all servers using a database repository you will have a configuration element in server.xml (or in some XML file included by server.xml) that looks something like this:

```
<batchPersistence jobStoreRef="BatchDatabaseStore" />
```

The jobStoreRef points to another element (a databaseStore) that contains information pointing to the actual database and tables to use for the repository. We've included that name (or at least the first 16 characters of it) in the SMF record. Is that name necessarily unique? Well...no. Our assumption was that in a production environment with multiple separate Job Repositories there would be some naming conventions common across the production server configuration that would require a unique jobStoreRef value pointing to different, commonly used/shared datasource definitions.

In Version 3 of the record, which increased the version of this section to 3, we introduced a new flag word, SM120CEI. The first byte of this flag word, SM120CEJ, contains a bit, SM120CEK, which reflects the value of the CVTZCBP field on the system writing the record.

Contents – Identification Section

Similar to the Identification section in the Type 30 record, the SMF 120-12 Identification section helps you to identify the job that was run in the server from the Subsystem section.

Field	*Description*	*Length*
SM120CBP	Version	4
SM120CBQ	Job Instance ID	8
SM120CBR	Job Execution ID	8
SM120CBS	Job Execution Number	8
SM120CBT	Step Execution ID	8
SM120CBU	Partition Number	8
SM120CBV	Job Name	32
SM120CBW	Application Name	32
SM120CBX	XML Name	32
SM120CBY	Step Name	32
SM120CBZ	Split Name	32
SM120CCA	Flow Name	32
SM120CCB	Create Time	8
SM120CCC	Start Time	8
SM120CCD	End Time	8
SM120CCE	Submitter	32
SM120CCF	Submitter JES Jobname	8
SM120CCG	Submitter JES JobId	8
SM120CCH	Dispatch TCB Address	4
SM120CCI	Dispatch TCB TTOKEN	16
SM120CDY	Flags (V2)	8
SM120CDZ	Step start limit (V2)	4

SM120CEA	Chunk Step Checkpoint Policy (V2)	4
SM120CEB	Chunk Step Item Count (V2)	4
SM120CEC	Chunk Step Time Limit (V2)	4
SM120CED	Chunk Step Skip Limit (V2)	4
SM120CEE	Chunk Step Retry Limit (V2)	4

As always, we begin with the version section number. After that we have a set of identifiers. Every job you submit has a unique job instance. A job instance has a unique identifier. When the submitted job actually runs, it creates a job execution. Each execution also has a unique identifier. If a job execution fails, you might restart it. If you restart a job, then it is a new execution of the same job instance. SMF records for the original job and the restart will have the same job instance identifier but different job execution identifiers.

The job execution number (not the execution ID) always starts with zero for any job instance. The first restart of the job (if there is one) will be execution number one, increasing each time the job instance is restarted. This field is only set for a job-ended record.

The step execution ID is another unique identifier assigned to the execution of a step. Note that a job-ended record won't have a step execution ID. Only step-ended and partition-ended records fill in this field.

I've been saying that these identifiers are unique. That's always a word that requires some clarification: unique within what scope? The identifiers are generated by the database underlying the Job Repository in use by the server. Therefore, the identifiers are unique within the set of servers that are sharing a set of Job Repository tables. Servers using an in-memory repository will generate identifiers that are unique within only that server. Sets of unrelated servers using different tables for their Job Repositories may generate the same identifiers. You'll need to be aware of what server wrote the record and your topology to be able to use these 'unique' identifiers safely. See the discussion about this in the Subsystem section just above (if you're skipping around).

The partition number field is intended to identify partitions for a partition step. So it will be filled in for a partition-end record, but no other record types.

The next field is the job name. This is taken from the 'id' tag for the job element in the JSL that defines the job being executed. This field and the ones that follow can actually be very long strings. We will truncate them to 32 characters to get them to fit in a reasonably sized SMF record.

The job-ended record gives us two more strings that help identify the job: the application name and the XML file name if the XML came from within the application. When a job is submitted one of the required parameters is the name of the application containing the batch job. This is probably the .war file name, but it can be other things depending on how the application is packaged. The batch container uses this name to set up the classloader so we can find all the Java code that is part of the batch job.

We also need to know what JSL you want to use to run the job. If the JSL is packaged with the application then we need to know the .xml file name containing the JSL. If you are using the REST interface or the z/OS native client to submit the job then you can also supply the JSL directly with the rest of the parameters. If you use JSL contained in the application package, then the name of the JSL file is included in the job-ended SMF record here.

For a step-ended record we will pick up the name of the step from the 'id' tag in the step element in the JSL and include it in the SMF record. We also include the step name in a partition-ended record.

In a flow-ended record we will fill in the split name and flow name fields.

Next we have some time stamps. The create time stamp will be set in a job-ended record to let you know when the job got created (essentially the time it got submitted).

> For a step-ended record this field contains the create time for the job since the step doesn't get 'created'. This is a change in Version 2 of the record. In Version 1 the create time was not set for a step-ended record.

The next two fields are the start/end times for the job or the step depending on whether this is a job-ended or step-ended record. Note that all these timestamps are in milliseconds since 1970 because that is the format used by Java (and not the STCK format you might expect in an SMF record).

> For a flow-ended record the timestamps relate to the flow itself. For a decider-end record the create time is the create time for the job, while the start and end times are for the decider itself. For a partition-end record the create time is when the partition plan is created. The start and end times are the times for the execution of the individual partition.

The job-ended record also gives you some information about who submitted the job. The first field is the userid that was used to submit the job. The next two fields identify the job that ran the native Command Line Interface (batchManagerZos) to submit the job. These two fields are picked up from the JSAB of the CLI from `JSABJBNM` and `JSABJBID` and placed in 'magic' job parameters (`com.ibm.ws.batch.submitter.jobName` and `com.ibm.ws.batch.submitter.jobId`). These fields will help you correlate the Liberty Batch job with the JCL job that submitted it, if that's something you want to do. If the JCL job that ran the CLI is submitted by an Enterprise Scheduler and you have chargeback processing already associated with that job, you may want to tie those records together with the matching SMF 120-12 records for the Liberty Batch job.

Finally, the identification section reports the thread in the server that executed the job. We provide both the TCB address and the TTOKEN of that TCB. The TTOKEN value is unique within the IPL of the system, whereas TCB addresses get reused. The TCB address is shorter and easier to work with, but depending on what you are doing you might need the more unique value.

> In Version 2 of the record we added some more things to the bottom of the section. These fields just report values taken from the JSL that defines the job. We start with some flags. Only the first three bits are defined in Version 2 (counting from the left of the 8 bytes we reserved for flags). The flags indicate whether the job is restartable, for a step-ended record if the will be run again if the job is restarted, even if the step completed, and for a step-ended record, if the step is a partitioned step.
>
> The step-ended record also fills in some more new fields that tell you about the JSL for the step. Most of these are self explanatory except checkpoint policy. If you are using a custom policy then this field is a '1', otherwise it is a zero.
>
> Why include all this stuff from the JSL in the SMF record? I thought perhaps it might be useful to enforce some standards (no chunk step time limits over one hour!) or possibly to help find applications doing certain things (any applications that have customer checkpoint policies), or possibly diagnostic (the checkpoint item count changed from 10,000 to 10, perhaps the more frequent checkpointing explains an increase in CPU usage).

Contents – Completion Section

The Completion Section is included in both the job-ended and step-ended records and contains information about how the job or step completed and what went on while it ran.

Field	*Description*	*Length*
SM120CCJ	Version	4
SM120CCK	Batch Status	4
SM120CCL	Exit Status	128
SM120CCM	Flags	8
SM120CCN	Partition Plan	4
SM120CCO	Partition Count	4
SM120CCP	Read Count	8
SM120CCQ	Write Count	8
SM120CCR	Commit Count	8
SM120CCS	Rollback Count	8
SM120CCT	Read Skip Count	8
SM120CCU	Process Skip Count	8
SM120CCV	Filter Count	8
SM120CCW	Write Skip Count	8

As always, we begin with a version. The version is followed by the batch status value. Batch status is an enumeration defined by the JSR-352 specification and includes values such as STOPPED, FAILED, and COMPLETED.

Part of the JSR-352 specification allows batch applications to set an Exit Status for a step or for the job itself. You can think of the Exit Status as the completion code for the job or step. Unlike a traditional JCL completion code, an Exit Status is a String. It can be as long or short as needed and could contain an error message, a huge JSON string, or simply a number (0, 4, 8, etc). It is up to the application developer. For SMF we will take the first 128 bytes of the Exit Status and drop it here. Unlike other string values in the SMF record which are in EBCDIC, we have no idea what might be in the Exit Status string. Conversion to EBCDIC might mangle it completely so expect the string to be in ASCII unless your application is using characters outside that range (e.g Cyrillic).

After the Exit Status is a double word field reserved for flags.

Next are fields for the partition plan and partition count. These are only filled in for a step-end record. The plan is how many partitions were planned to execute and the count is the number that actually ran. Normally these would be the same, but in error cases they might not be. For a non-partitioned step these are set to negative one (-1) to avoid colliding with a zero for partition count that would mean no partitions ran.

The rest of the record consists of counts taken from the JSR-352 specified Metrics. These counts are only set for step-ended record and only apply for chunk-type steps.

Contents – Processor Section

In this section we get information about how much processor was used in the execution of the job or the step.

Field	*Description*	*Length*
SM120CCX	Version	4
SM120CCY	Total CPU Start	8
SM120CCZ	Total CPU End	8

SM120CDA	Time on CP Start	8
SM120CDB	Time on CP End	8
SM120CDC	Offload CPU Start	8
SM120CDD	Offload CPU End	8
SM120CDE	Offload on CP Start	8
SM120CDF	Offload on CP End	8

All the values in this section are obtained by calling the z/OS TIMEUSED API. That API may or may not be able to give us back all the values we'd like, depending on whether or not something called the Extract CPU Time (ECT) hardware facility is available. If not, then we can just get the total CPU value. If the ECT is available then we can also get back information about GP vs. offload CP usage information. CPU times are returned in TOD format. Divide by 4096 to get microseconds.

Whatever information we have, it is obtained by calling TIMEUSED at the start and end of the job and of the step. This provides us with the amount of CPU used by the dispatch thread at the start and end of the job or step. In the SMF 120-12 we report both values. Why? Does anybody care about the actual values? Don't you really just want the CPU used which you could easily get by subtracting the starting value from the ending value? Yes indeed. However, historically a large percentage of the APARs WebSphere has taken against our SMF records have involved situations where we had absolute values like this and did math for you to calculate the values reported in the SMF record. Therefore we decided in this case to just give you the start and end values and let you do the math yourself. One less thing for us to get wrong.

What do all these values mean? The first pair is the total CPU used of all types. The second pair is the amount of GP processor used. The third pair is the amount of specialty (or offload) processor used (e.g. zAAP or zAAP on zIIP). And the fourth pair is the amount of offloadable processing that actually ran on a GP processor. This last pair might have non-zero values if, for example, you have IIPHONORPRIORITY set to YES which can result in zIIPable processing overflowing onto the GP processors.

You should note that in the case of a job containing a split/flow the steps executed as part of the flows run on different threads than the main job thread. In this case the CPU used by the steps as reported in the step-end records will add up to more than the CPU used as reported in the job-end record since the job-end record only reports the CPU on the main job dispatch thread at the start and end of the job. To get total CPU for the job you need to include CPU from all the steps regardless which thread they executed on.

> The flow-ended record does not contain a processor section because all of the CPU used in the steps in the flow are reported in the appropriate step-ended records. Reporting CPU for the complete flow would be double-counting.
>
> Similarly, if a job includes a partitioned step the step-ended record will only include CPU used by the parts of step execution than ran on the main step thread (e.g. PartitionMapper etc). The actual partitions run on separate threads and that CPU usage is reported in the partition-end record for each partition.
>
> It is somewhat counter-intuitive, but the safest way to calculate the CPU used by a job may be to ignore the CPU reported in the job-end record and instead find all the step-end and partition-end records for the job and add those up. But be careful. The job-ended record reports CPU used by the dispatch thread for the entire job. The step-ended records for steps that ran on that thread only include CPU used in the step itself. So it is possible (likely even) that the job-ended record will report more CPU used than the sum of the step-ended CPU records for steps that ran on the main thread.

> What to do? You can use the dispatch TCB address (SM120CCH) to determine the main thread for the job. Use the job-ended record to get CPU from the main thread. Then go find the step-ended records for any step in this job that didn't run on the main thread. And also add in any partition-ended records you find for this job since they never run on the main thread. That should be the closest you can get to CPU used by the job.
>
> And remember, you are finding all these different records using the job execution id, which is only unique within the scope of the Job Repository (see the discussion about that in the subsystem section earlier).

Contents – Accounting Section

This section contains accounting information similar to what is provided in a Type 30 record lifted from the ACCT parameter in JCL.

Field	*Description*	*Length*
SM120CDG	Version	4
SM120CDH	Accounting String Length	4
SM120CDI	Accounting String	128

This is a repeating section. The number of sections you have depends on what the submitter of the job provided in the job parameters. We recognize another 'magic' job parameter called com.ibm.ws.batch.accountingString. If that parameter is not set, then this section will not occur in the SMF record. If the parameter is set then we consider it to be a list of comma separated strings. Each string is parsed out and set into one instance of this section, up to 128 characters. The length field tells you how much of that 128 character field is actually filled in.

For example, if you set the parameter to “12345,DEPT ABC” then you would get two instances of the Accounting Section. One instance would set the string to “12345” and the other instance would set it to “DEPT ABC”.

You may wish to set this parameter to match ACCT information in the JCL job used to drive the batchManagerZos client that submits the Liberty Batch job. Post processing could then accumulate chargeback information using the same accounting information for both traditional JCL and Liberty Batch jobs.

Contents – USS Section

This section contains information about the server and the thread where the job ran from a Unix System Services perspective. Logically this information probably could have just gone in the Subsystem and Identification sections. However, for the SMF Type 30 this information is kept in a separate section and we thought we'd do the same here.

Field	*Description*	*Length*
SM120CDJ	Version	4
SM120CDK	Server PID	4
SM120CDL	Dispatch Thread ID	8
SM120CDM	Dispatch Java Thread ID	8
SM120CEF	Submitter's UID (V2)	4

SM120CEG	Submitter's GID (V2)	4

This is pretty easy. We give you the process id (PID) of the Liberty server where the job ran. We give you the thread ID where the job or step ran within the process. And finally we ask Java for the thread id by calling Thread.getCurrent.getThreadId() and report the result here.

In the version 2 record we also include the UID and GID of the submitting user in the record.

The Reference Section

In version one of the record there was an always-empty triplet at the front for something called the Reference Section. In version two we actually filled it in.

Field	*Description*	*Length*
SM120CDN	Version	4
SM120CDO	Type	4
SM120CDP	Reference String Length	4
SM120CDQ	Reference Name	128

This section only appears for a step-ended record and you can have more than one of them depending on the type of step. For a batchlet you will just get one instance with a type value of '5' and the name will contain the reference value of the batchlet (probably the package and class name of the batchlet implementation class, but not necessarily). For a chunk step you get three instances of the section, one each for the reader, processor, and writer.

If the step is partitioned you will also get sections for the various partition related code (e.g. partitionMapper) if you have them.

This is intended to be roughly equivalent to an SMF Type 30 telling you the name of the load module that ran for the step. A step in JSR-352 can be a bit more complicated.

The User Data Section

If you look closely at the mapping of the SMF 120-12 record you will notice that the triplet section at the top of the record contains a triplet for a User Data section. Any actual SMF data you get for Liberty Batch jobs will have the 'count' field in the that triplet set to zero. That's because this section doesn't actually exist. We may add it in the future in which case the counts will become non-zero if the data is present. SMF processing code can always check the count field for a zero (which you should be doing anyway) and just do nothing in that case.

Document change history

Check the date in the footer of the document for the version of the document.

September 26, 2016	Initial Version
September 29, 2016	Add document number
May 1, 2017	Corrected batch SMF feature name
April 9, 2019	Increased record version to 3, increased Subsystem section version to 3, introduced new flag word, SM120CEI, and flag SM120CEK

End of WP102668