Exploiting New JES2 Interfaces

SHARE, Winter 2004

Project: JES2

Session: 2664

Tom Wasik
JES2 Development
wasik@us.ibm.com

The presentation discusses how to use the newer interfaces to JES2 with some assembler coding examples. It is intended for programmers that may wish to exploit the interface or to understand what is possible using these interfaces. What are "new" interfaces? Well new is relative. In my way of thinking, new is anything that was introduced after I started working with the product (SP 1.3.6). But for this presentation, I will be talking primarily about SAPI, Extended status, and Spool browse with a few others thrown in.

Overview

This presentation will cover the following:

- What is the SSI and how is it used
- Extended status SSI request
- SAPI SSI requests
- SPOOL Browse
- Other SSIs
  - Who AM I SSI
  - Notify SSI
  - SPOOL read SSI

This presentation will discuss various interfaces to interact with JES2. Primarily, these interfaces use the Subsystem interface (SSI). I will start with a quick overview of the SSI and then go into the various SSI calls JES2 supports and what can be done with them. SPOOL Browse is the only interface that we will be discussing that does not use the SSI to interface to JES2.
The examples that will be provided show how to use the various interface but do not do anything useful with the data provided. Also the methods of obtaining input is often using simple WTORs. In that sense, these are intended as examples of how to use the interface, rather than complete programs that an installation would want to use.

What is the SSI

- The SSI is an MVS interface to "Subsystems"
  - Used as a hook to give info to subsystems
    - WTO, CMDs, EOT, EOM, etc.
  - Used as a way to request functions
    - PSO, SAPI, Extended Status
  - Each SSI has a number and an SSOB extension
  - Subsystem identifies what it supports
  - Caller can specify subsys to process request
    - Default, Specific, All

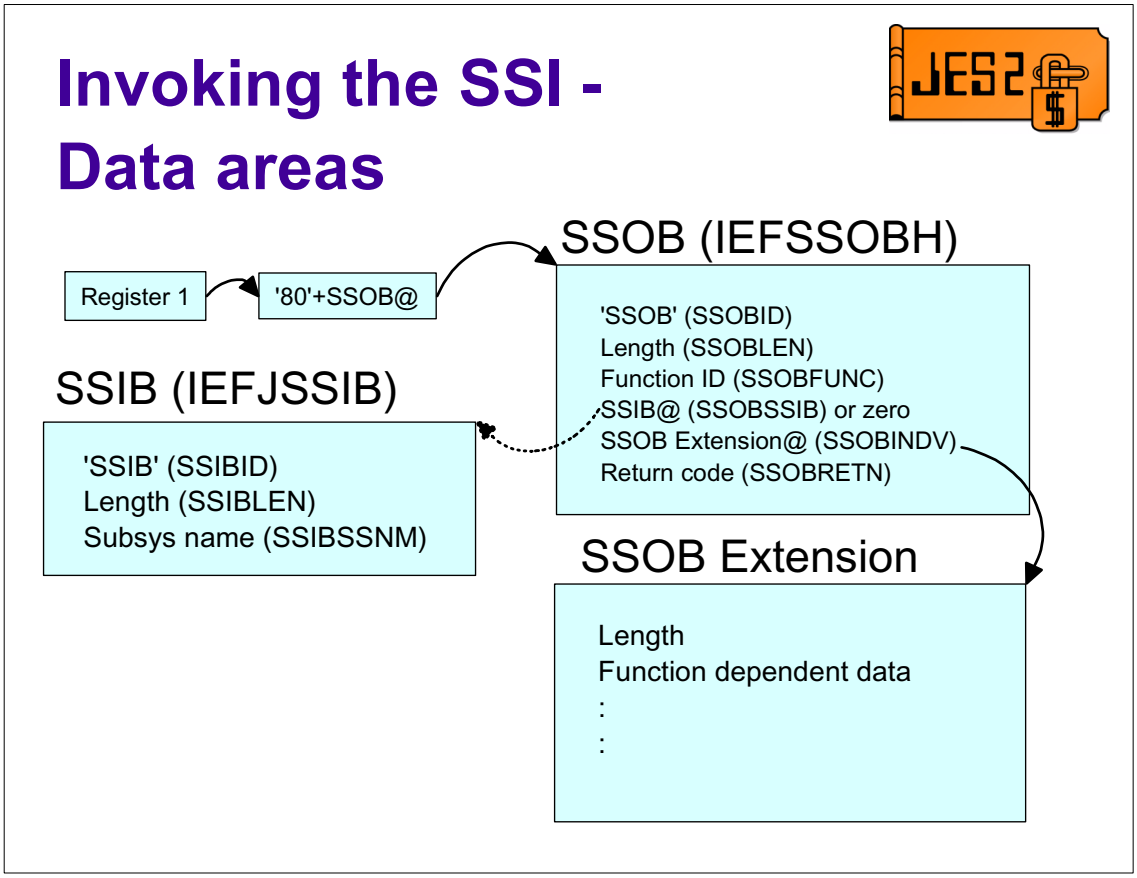
The SSI is an MVS interface to subsystems. A subsystem in this context is defined as any program that responds to SSI requests. JES2 and JES3 are two of the major users of the SSI interface. The SSI functions as both an hook that provides information to the subsystems when certain events occur, as well as a way to request information/services from a subsystems. WTO, command, End of task, End of Memory are all examples of SSIs that are invoked by MVS to tell a subsystem that something has happened. These SSIs are intended to only be issued by MVS and listened to by subsystems. PSO, SAPI, Extended Status are all examples of SSIs that are invoked by applications that are requesting services from a subsystem.
Each SSI has associated with it a number and an SSOB extension. The numbers (normally stated in decimal) ensures that the proper function is requested. The SSOB extension is where the parameters for the specific SSI are defined.
Each subsystem must identify to MVS what SSI numbers (function codes) it supports. The next chart lists the function that JES2 supports (for use by applications).
SSI calls can be directed to the default subsystem (the one the application was started under), a specific subsystem, or all subsystems. Sending a request to all subsystems is called a broadcast SSI. Only certain SSIs support being broadcast. The only SSI available to applications that can be broadcast is the extended status SSI.

What is the SSI *(cont...)*

The SSI calls (that applications can use) which JES2 supports are:

Number	Symbol	Macro	Description
1	SSOBSOUT	IEFSSSO	Process SYSOUT
2	SSOBCANC	IEFSSCS	Job cancel
3	SSOBSTAT	IEFSSCS	Job status
11	SSOBUSER	IEFSSUS	Destination validation/conversion
20	SSOBRQST	IEFSSRR	Request job ID
21	SSOBRTRN	IEFSSRR	Return job ID
54	SSOBSSVI	IEFSSVI	Subsystem Information
70	SSOBSFS	IAZSSSF	SJF spool services
71	SSOBSSJI	IAZSSJI	Job information
75	SSOBSSNU	IAZSSNU	User notification
79	SSOBSOU2	IAZSSS2	SYSOUT API (SAPI)
80	SSOBESTA	IAZSSST	Enhanced status information

This table lists the SSI request that are available to applications that are supported by JES2. Newer SSIs have the higher numbers. Some of these SSIs are documented in the *z/OS V1R5.0 MVS Using the Subsystem Interface* book (SA22-7642-03). However, most of the newer SSIs have fairly complete documentation in their SSOB extensions (Macro column in the table).



The major data areas that must be filled in to invoke the SSI are the SSOB and the SSOB extension. If you want to direct the request to a specific subsystem, then you can also pass an SSIB on the request. The SSOB extension that is used will depend on the function ID (SSI number) being used.

Invoking the SSI - Code


```
        USING SSOB,MYSSOB         Establish SSOB addressability
        SPACE 1
        XC    MYSSOB,MYSSOB       Zero SSOB area
        LA    R6,MYSSOB           Get address of SSOB
        SPACE 1
        MVC   SSOBID,=C'SSOB'     Set SSOB eyecatcher
        MVC   SSOBLEN,=Y(SSOBHSIZ)  Set length of SSOB header
        MVC   SSOBFUNC,=Y(SSOBSSxx)  Set function code
        MVC   SSOBSSIB,=F'0'      Use LOJ SSIB
        LA    R0,SSOB+SSOBHSIZ    Point to SSOB extension
        ST    R0,SSOBINDV         Point base to extension
        SPACE 1
        USING SSxxxxx,SSOB+SSOBHSIZ  SSOB extension addr'blty
        SPACE 1
* Code to set up SSOB extension goes here
        SPACE 1
        LA    R6,MYSSOB           Point to SSOB
        O     R6,=X'80000000'     Set HI BIT to indicate last
        ST    R6,PARMPTR          Save SSOB address in parm
        LA    R1,PARMPTR          Get pointer to SSOB
        SPACE 1
        IEFSSREQ                  Invoke the SSI
        SPACE 1
        LTR   R15,R15             If this is nonzero
        JNZ   SSREQERR             we're in big trouble
        CLC   SSOBRETN,=A(0)        Is there an error?
        JH    SSOBERR                 Yes, process error
```

This is the basic code needed to invoke any SSI request. This code sends the request to the subsystem associated with the address space (uses the life of job SSIB). This SSIB points to the subsystem that started the address space. If the address space was started under the master subsystem (does not have a job structure in JES2 or used request job id), then the request will go to the MSTR subsystem. If it was started under JES2 (has a job structure that is not from request jobid) then the request will go to the JES2 instance that started the address space.

Notice that after the call, there are 2 return codes being checked. The R15 value after the call to IEFSSREQ is a function independent return code defined in IEFSSOBH. These return code are often not set by the subsystem itself but rather by the IEFSSREQ logic. The SSOBRETN is a function dependent return code that is defined in the individual SSOB extensions. These are always set by the subsystems. Often there will be a third return code (or a reason code) in the SSOB extension itself to further identify the cause of an error.

Warnings

- Most SSIs require caller to be APF authorized
  - Ensure code is properly tested
- Examples are just that
  - Not intended for production
  - Needs validation of input
  - May have security issues
  - Need to add recovery

Before you start writing code or looking at the examples, some words of warning.
SSIs requests generally require their callers to be authorized. Since these are authorized interfaces care must be taken to ensure that the code will not create an integrity exposure or cause a system failure. Careful review of the code and lots of testing in a test environment will help minimize the risk.
The examples that are being presented are just that, examples. They are not intended as production code. In many cases, the examples you are looking at were written in development to exercise the JES2 SSI supporting code. Not much care was given to usability, recovery, validating input, error checking, etc.. The examples are intended to give you something that works that you can use as a basis for developing your own applications. Some examples may function exactly as you want them to. That still does not mean they are ready for a production environment.

Extended Status SSI

- Obtain JOB and SYSOUT information
  - Data in JES2 checkpoint returned
    - CKPT version used
  - SSI function 80 (IAZSSST mapping macro)
  - 3 call types
    - Get job data
    - Get SYSOUT and JOB data
    - Release memory
  - Filters control data returned
  - Supports directed SSIs and broadcast

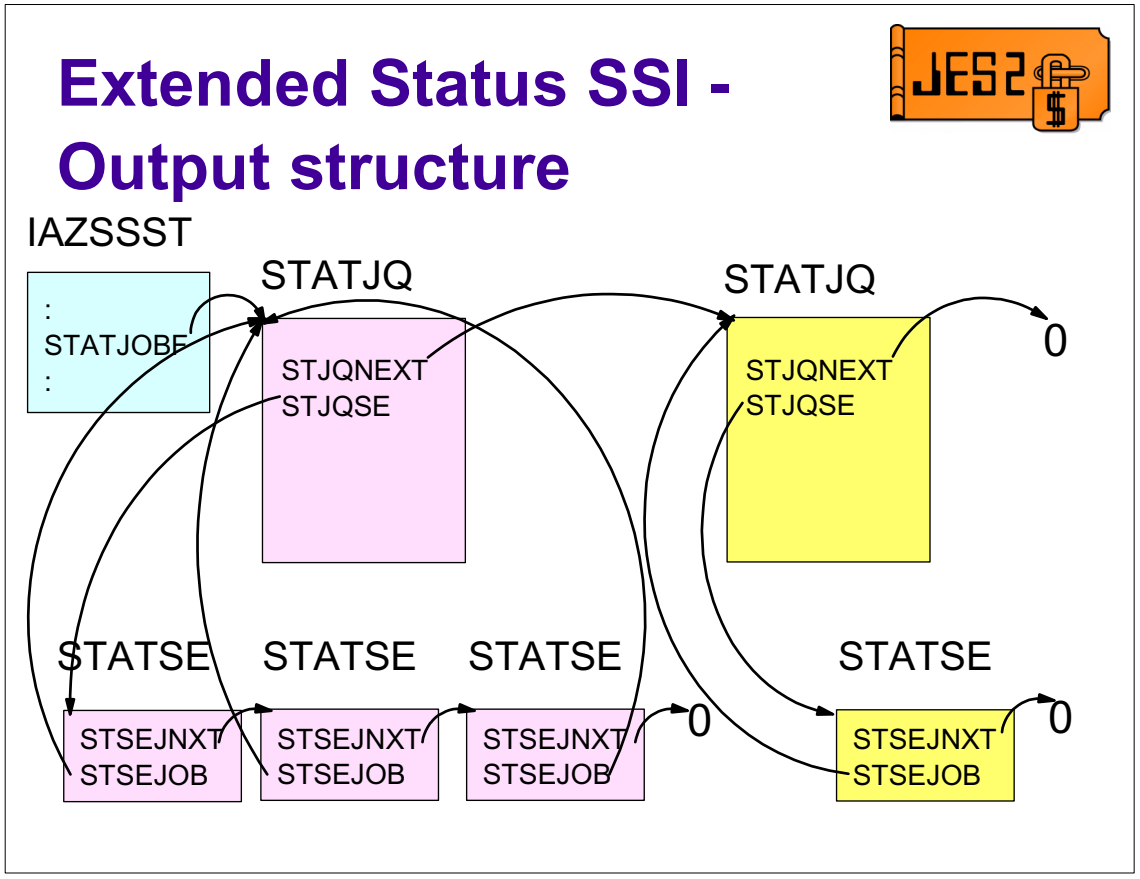
The extended status SSI obtains job and SYSOUT information from the JES2 checkpoint using a checkpoint version (to ensure the consistency of the data returned). Only data that is in the JES2 checkpoint is available to be returned via this SSI. Processing in this SSI does not affect the processing in the JES2 address space (since all the code runs in the requesting address space).
There are 3 functions supported, get JOB data, get SYSOUT (and job) data, and return storage. A very typical characteristic of many of the newer SSIs is a pair of calls, one to obtain data in storage obtained by the SSI, and then a memory management call to release the storage. You will see this on many of the SSI calls.
The requester can filter the data returned based on a wide range of JOB as well as SYSOUT filters.
The SSI supports both directed an broadcast requests. What this means is that you do not have to be running under the target subsystem to make this request. Since this SSI also supports a broadcast request, you can ask all subsystems (all JES subsystems) on a system to return data in one call.

Extended Status SSI - SSOB structure

- The IAZSSST (SSOB extension) is structured as follows:

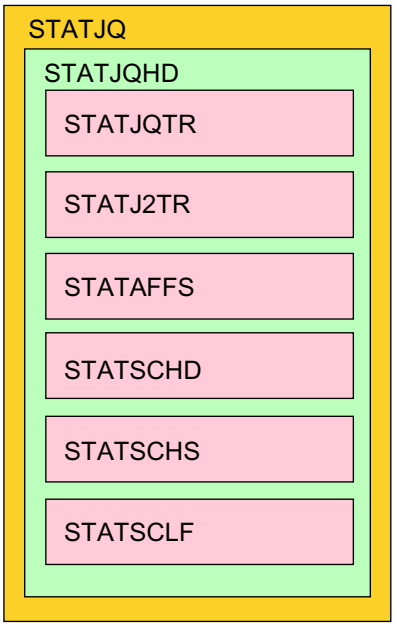
Standard SSOB stuff (Length, eyecatcher, version)
Additional error reason codes (STATREAS, STATREA2)
Function requested (STATTYPE)
Input filter bit masks (STATSELx, STATSSLx)
Input JOB level filter fields
Output area pointers and counts
Input SYSOUT level filter fields

The SSOB extension is mapped by IAZSSST. The extension is made up of a number of sections, each representing a different function. Filtering is accomplished by setting a bit in the appropriate byte and then setting the corresponding field to the value to filter on. Output areas are chained into the SSOB extension.



The output areas returned by extended status are pointed to by STATJOBF in the SSOB extension (IAZSSST). There are 2 types of output areas. STATJQs represent a job (JQE). For every job which matched the filter criteria, a STATJQ is built. STATSE represent an output group (JOE). The STATSEs are chained out of the STATJQ (so you if you ask for SYSOUT information, you will always get STATJQs too). The STATSEs point back to the STATJQs that own them.

Extended Status SSI - STATJQ

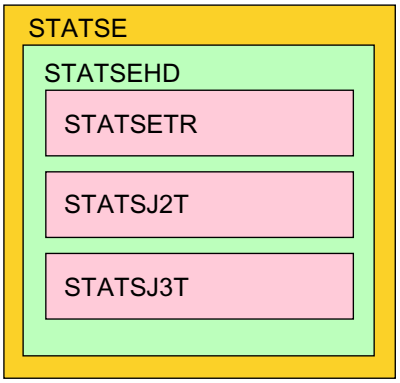


Section in the STATJQ
- STATJQ - represents job
- STATJQHD - describes output areas
- STATJQTR - Job Queue Element terse section
- STATJ2TR - JQE JES2 terse section
- STATAFFS - JQE member affinity section
- STATSCHD - JQE scheduling section
- STATSCHS - JQE SCHENV affinity section
- STATSCLF - JQE SECLABEL affinity section

Sections work like NJE header sections
- Each section has a length, ID, and modifier
- Use lengths to step through sections
- STATJQHD has overall length to end of area
- NEVER USE ASSEMBLER LENGTHS EQUs
  - STATJQHD is only exception
- Not all sections are always present

The STATJQ is composed of a number of sections. The high level DSECT (STATJQ) has the pointers to the next STATJQ and to any STATSEs. The length of the STATJQ header is also in the area. Add the length field to the STATJQ and you point to the STATJQHD. This is a header for the remaining fields. It has the overall length of the remaining areas. This length is used to determine when you have reached the end of the variable sections. You add the STATJQHD length equate (STHDSIZE) to the address of the STATJQHD to get the first variable section. Each variable section starts with a 2 byte length (STxxLEN), a 1 byte ID fields (STxxTYPE) and a 1 byte modifier (STxxMOD). When scanning for or identifying a section, ensure you check both the type AND modifier to determine what section this is. To get to the next section, add the STxxLEN field to the current section pointer. Not all sections are present for all jobs. Also, maintenance or a new release can add new section types or modifiers to existing types. Ensure you application can handle unknown types.

Extended Status SSI - STATSE

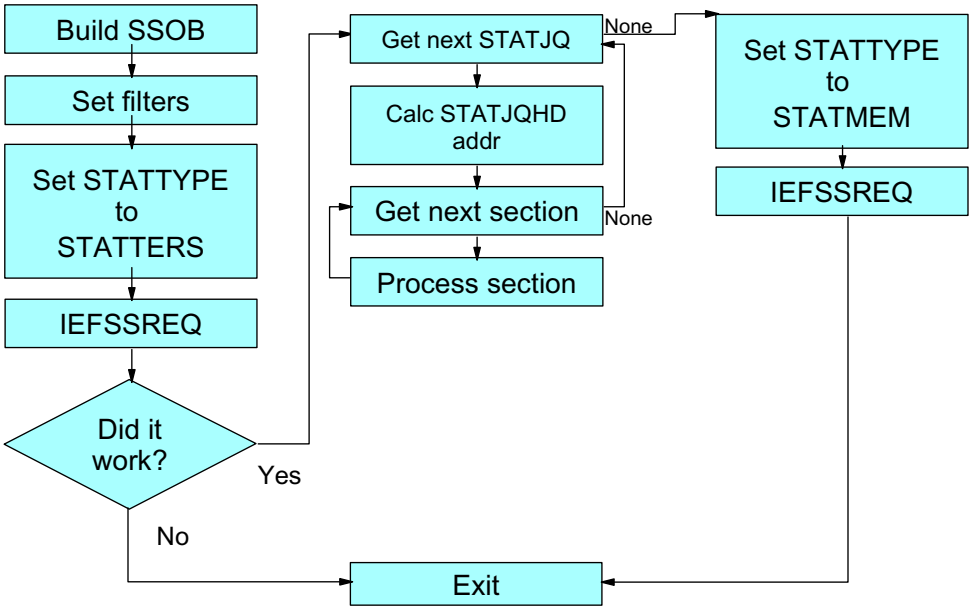


Section in the STATSE
- STATSE - represents SYSOUT group (JOE)
- STATSEHD - describes output areas
- STATSETR - SYSOUT element terse section
- STATSJ2T - SYSOUT JES2 terse section
- STATSJ3T - SYSOUT JES3 terse section

Sections work like NJE header sections
- Each section has a length, ID, and modifier
- Use lengths to step through sections
- STATSEHD has overall length to end of area
- NEVER USE ASSEMBLER LENGTHS EQUs
  - STATSEHD is only exception
- Not all sections are always present

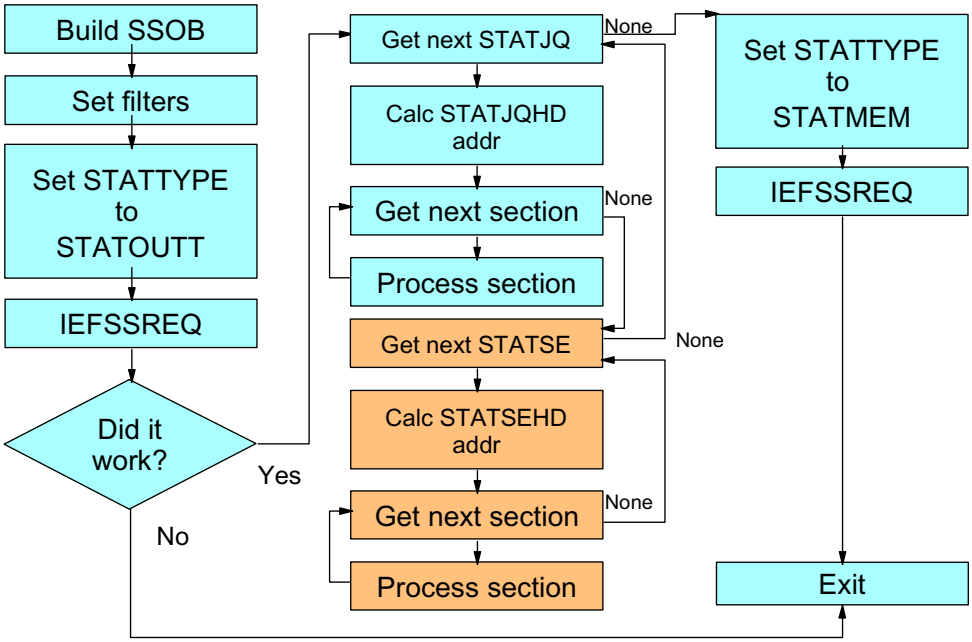
The STATSE is also composed of a number of sections. The high level DSECT (STATSE) has the pointers to the next STATSE and back to the job level STATJQ. The length of the STATSE header is also in the area. Add the length field to the STATSE and you point to the STATSEHD. This is a header for the remaining fields. It has the overall length of the remaining areas. This length is used to determine when you have reached the end of the variable sections. You add the STATSEHD length equate (STSHSIZE) to the address of the STATSEHD to get the first variable section. Each variable section starts with a 2 byte length (STxxLEN), a 1 byte ID fields (STxxTYPE) and a 1 byte modifier (STxxMOD). When scanning for or identifying a section, ensure you check both the type AND modifier to determine what section this is. To get to the next section, add the STxxLEN field to the current section pointer. Not all sections are present for all SYSOUT areas. Also, maintenance or a new release can add new section types or modifiers to existing types. Ensure you application can handle unknown types.

Extended Status SSI - Typical logic (JOB info)



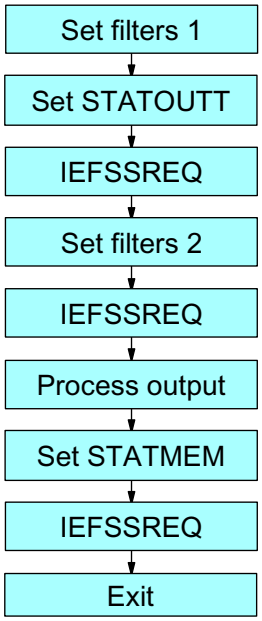
A typical logic flow to obtain and process job level information is pictured here. The application set up the SSOB and filters, invokes the SSI, then processes the output. Once all the elements have been processed, the storage is freed with another IEFSSREQ request. Note that once the SSOB is built and passed to the SSI, it is not updated (except for the STATTYPE). It is important that application pass the same SSOB into both SSI calls without updating the fields labeled as output fields.

Extended Status SSI - Typical logic (SYSOUT info)

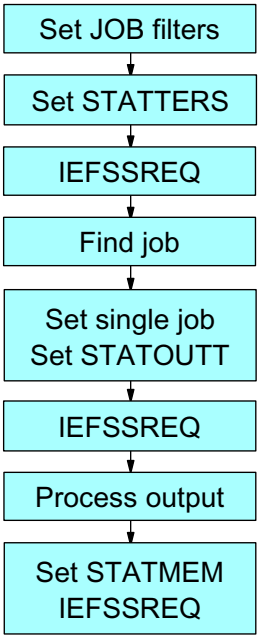


This flow adds SYSOUT information to the output returned. SYSOUT data areas are chained into the STATJQ data areas.

Extended Status SSI - Less typical flows



Multiple SSI requests with different filters can be made without freeing the output areas. This allows "OR" processing for filters. For example if you want information for SYSOUT that is class A or class B.



The first of multiple calls can request job information and then, select a job and add the SYSOUT information for one of the jobs (without a memory management call).

Multiple SSI calls can be combined to without making a memory management call. This achieves an "OR" function for filters. Furthermore, the first call can be a job call that obtains STATJQ data areas, a job can be selected from the returned data areas, then by setting a single job filter and STATOUTT, the SYSOUT information for that job will be added to the STATJQ data area already obtained.

Extended Status SSI - Other facts

- You can set STATTERS and specify SYSOUT filters
  - Only STATJQs for jobs that have output that matches the filters are returned (no STATSEs returned)
- STATSEs have a JOE level token (STSTCTKN) that can be passed to SAPI to process the output
- STJ2SPOL can be passed to SPOOLIO SSI to read the JCT for the job

All filters can be used on all call types. That implies that SYSOUT filters can be used when job level information is being requested. When this is done, only jobs that have output that matches the filter will be returned (and only STATJQs are returned).
The STATSE contains a JOE token that can be passed on the SYSOUT API (SAPI) to select a specific output group for processing.
The STATJQ contains a SPOOL token that can be passed to the SPOOLIO SSI to read the JCT for a job.

Extended Status SSI - Processing routine

- If you need high performance and do not want to use as much memory, use the processing routine (STATRTN)
- Instead of returning data, it is passed to the processing routine.
  - For JOBs you get called for each STATJQ
  - For SYSOUT you get called for each STATSE
- Set STATRTN and STATRPRN before SSI
- STATPARM maps the R1 parameter list
- See STATSSST for more information

When using the extended status SSI to obtain information on a large number of jobs, you may want to reduce the amount of storage that is obtained. This is particularly true if you plan to copy the data obtained into your own work area and are concerned about having 2 copies of the data in storage at the same time. To address this, the extended status SSI supports a processing routine. This routine is called as each output element is generated. If the is a job level request, then it is called every time a STATJQ is built. If this is an SYSOUT level request, it is called for every STATSE that is built (the STATJQ is passed with the STATSE). The processing routine can then copy the data to their own work area and return to the SSI. A data area can be passed in the IAZSSST that is passed to the processing routine. Also, the processing routine can set a field that is passed between calls to the processing routine. The parameter list is mapped by STATPARM in the IAZSSST macro. Even though a processing routine is used and no data areas are passed back, a memory management call is still needed to delete internal work areas that were obtained.

Extended Status SSI - Program example

- STATUS2 is the example program
  - Authorized TSO command
  - "STATUS2 ?" gives a brief explanation of options
- Output is a hex display (dump format) of
  - SSOB and extension
  - All output areas returned (STATJQs and STATSEs)
- Useful to see format of data returned

The STATUS2 program is an example of how to use the extended status SSI. STATUS2 is an authorized TSO command that takes as input the various filters that are supported by extended status and builds and IAZSSST (SSOB) to invoke the SSI. The command syntax can be obtained by issuing "STATUS2 ?".
The output of the command is very primitive. The SSOB as well as any STATJQs and STATSEs are displayed in hex format with the EBCDIC translation (similar to what is seen in a dump). This provides an easy way to look at what is returned from the SSI call.

SYSOUT API

- Allows applications access to SYSOUT
  - SYSOUT must not be active (busy)
  - Can only access non-NJE routed SYSOUT
  - Similar to PSO but with more function
- SSI function 79 (IAZSSS2 mapping macro)
- 3 primary functions
  - PUT/GET - accesses individual data sets
  - Count - returns various counts from JOEs
  - Bulk Modify - Alters SYSOUT characteristics
- Filters control what JOEs are processed
- Address space must be known to JES

The SYSOUT API (SAPI) provides a rich set of services that provide information about SYSOUT on the system. It is intended as an enhancement to the PSO interface (which is no longer being enhanced). The primary function of SAPI is to act as a printer application. Since it is a printer application, it can only access non-NJE bound output that is not already busy on another device. However, it can access output that is held (JCL type held) as well as non-held. This is to be compatible with some PSO functions.
Other functions available via SAPI are a count request and a bulk modify request. Count requests simply look at all the JOEs that match a selection criteria and count the number of elements that match and accumulate the various counts (e.g. lines, pages, etc.). This is useful if you are trying to determine how much output is waiting to be processed.
Bulk modify is similar to the PSO group request. It allows you to make changes to the characteristics of a set of JOEs with one SAPI call. With bulk modify you can alter the SYSOUT's class, destination, or release SYSOUT from held status. In addition you can delete SYSOUT.
What SYSOUT data sets to process is controlled by various filters specified in the IAZSSS2.
The SAPI SSI requires that the requesting address space have a job structure associated with the target JES2. Only authorized callers are allowed.

SYSOUT API - SSOB (IAZSSS2)

- IAZSSS2 identifies:
  - Function to be performed
  - Filters to select output to be processed
  - Output information associated with request
- One request may need multiple SSI calls
  - A *thread* is a set of related SSIs
  - First call clears SSS2JEST
  - Last call sets SSS2CTRL on
  - If done and SSS2JEST is non-zero, SSS2CTRL call is needed
  - Errors can cause JES2 to terminate thread

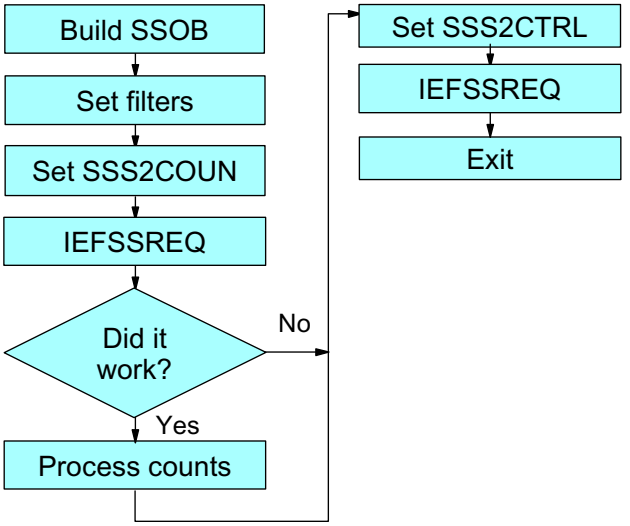
The IAZSSS2 contains both input and output fields. The input includes the function to be performed and a set of filters to select which SYSOUT data sets are to be processed. The output information includes the job and SYSOUT characteristics, pointers needed to allocate data sets, SWB information, and other fields that may be needed to process the SYSOUT. Input fields which are valid and output fields that are returned are based on the function requested. PUT/GET calls return the most data, bulk modify returns just a success or failure indicator.
When a SAPI request is made, an environment is created to process this request. The assumption is that multiple requests will be made using this same environment. The set of requests that share this environment is called a *thread.* A new thread is started when a SAPI request is made with SSS2JEST set to zero. This thread is maintained until either the task that created the thread terminates, or a SAPI call is made with SSS2CTRL being set on. One task can have multiple threads associated with it. Threads can be passed between tasks (however it is always owned by the creating task). Under certain error scenarios, a thread can be terminated by JES2, but normally, it takes a successful SAPI request with SSS2CTRL set. When SSS2CTRL is set, the function requested is not performed, except if a JOE is currently assigned to the thread (from a previous PUT/GET call). Then normal put processing will occur. Setting SSS2CTRL on a call with SSS2JEST set to zero does nothing.

SYSOUT API - Security

- PUT/GET and Bulk modify calls perform JESSPOOL RACF checks
  - One check per data set (NOT at JOE level)
  - Data sets (PDDBs) that fail check are ignored
  - Can cause one JOE to be broken into multiple
    - 5 PDDB JOE, requester cannot access 3rd PDDB
    - Bulk modify request to change class
    - Result 2 JOES, #1 has PDDBs 1, 2, 4, 5; #2 has PDDB 3.
- No RACF checks for count request

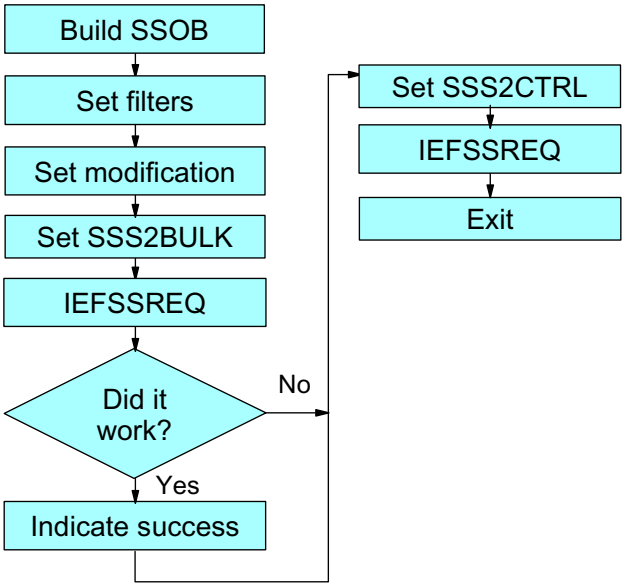
SAPI performs RACF checks to ensure that the requester has access to the SYSOUT. The checks are the standard JESSPOOL class checks made elsewhere in JES2. UPDATE access is always required. The checks are made at the SYSOUT data set level (PDDB) not the JOE level. This implies that an application may be able to access some data sets in a JOE but not others. Data sets which the application does not have access to are ignored. The application has no way to know that there are data set which it has no access to. Security failures can cause a single JOE to be split into multiple JOEs. For example, if there is a 5 PDDB JOE in which the application has access to all but PDDB 3, then a bulk modify request to change the class of this data set can only effect PDDBs 1, 2, 4, and 5. So when the request completes, there will be 2 JOEs, one with PDDBs 1, 2, 4, and 5 and one with just PDDB 3.
The security checks apply to PUT/GET and bulk modify request. They do not apply to count requests.

SYSOUT API - Typical logic (Count)



A count request starts by building the SSOB (IAZSSS2) and setting the selection criteria for the JOEs to be counted. The SSOB is set to be a SSS2TYPE of SSS2COUN. The IEFSSREQ is then issued and the results checked. Return codes come back in R15 (IEFSSREQ return code), SSOBRETN (SAPI return code) and SSS2REAS (SAPI reason code). If the request worked, the counts are stored in SSS2LNCT (line count), SSS2PGCT (page count) and SSS2CDS (JOE count).
Once the call completes, the thread that was created is still active. If the task is terminated, then the thread will be terminated. However, if you want to terminate the thread before the task terminates, another call must be made with SSS2CTRL set on. This tells JES to terminate the thread without performing the count function that is still set in SSS2TYPE.

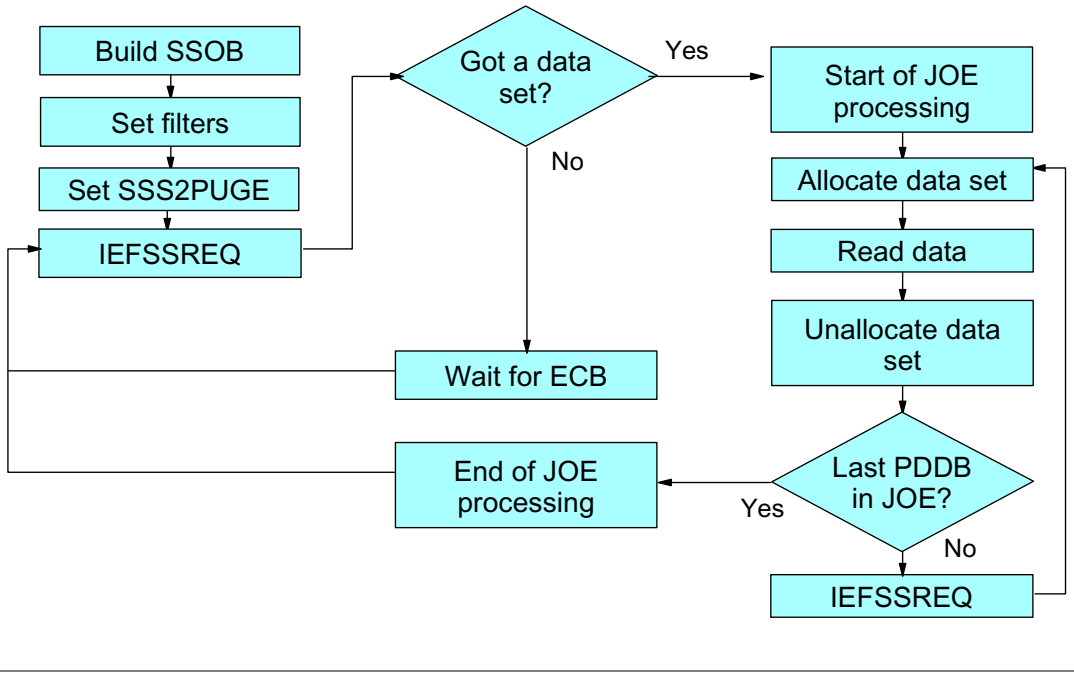
SYSOUT API - Typical logic (Bulk Modify)



A bulk modify request is very similar to a count request. It starts by building the SSOB (IAZSSS2) and setting the selection criteria for the JOEs to be modified. In addition, fields that indicate the type of modification are also set in SSS2UFLG. The SSOB is set to be a SSS2TYPE of SSS2BULK. The IEFSSREQ is then issued and the results checked. Return codes come back in R15 (IEFSSREQ return code), SSOBRETN (SAPI return code) and SSS2REAS (SAPI reason code). If the request worked, the output groups have been modified.

Once the call completes, the thread that was created is still active. If the task is terminated, then the thread will be terminated. However, if you want to terminate the thread before the task terminates, another call must be made with SSS2CTRL set on. This tells JES to terminate the thread without performing the bulk modify function that is still set in SSS2TYPE.

SYSOUT API - Typical logic (PUT/GET)



This logic shows a typical application that selects JOEs that match some criteria, process the JOEs, then wait for more JOEs to become available. As illustrated, this is an unending process. A real application would have a command to halt the process or it would exit when no work was found.
The processing starts with setting the selection criteria, indicating it is a PUT/GET calls, then invoking the SAPI SSI. If no SYSOUT is returned, then the process waits for JES2 to post that work is available. If a SYSOUT data set was returned (a JOE), then processing starts with setup for the new JOE (Header processing, data set allocation, etc.). The first data set is allocated, opened, and read. Once it has been read and processed, the data set is closed and unallocated. If there are more data sets in this JOE, then another SAPI SSI call is made to dispose of the first data set and get then next data set. If the last data set for the JOE (SSS2DSL is on) was just processed, then cleanup processing for the JOE is done. Another SAPI SSI call is made and the last data set is disposed and a JOE is obtained and the processing continues.

SYSOUT API - Put/Get Advanced topics

- SYSOUT characteristics
  - Most JOE/PDDB fields returned
  - SWBs (TUs or SWA) available
  - NJE job and data set headers
  - RACF security token
- Disposition processing
  - Keep or "process" the data set
  - Hold the data set (system or JCL)
  - Change attributes
  - Don't show to thread/address space again

The output area in the IAZSSS2 contains all the traditional characteristics of the SYSOUT and owning job. These are the fields obtained from the JOE, JQE, JCT, and PDDB. In addition, the output SWBs (data from OUTPUT JCL statements) and the NJE job/data set headers are also made available (the NJE headers are only available if the job or output arrived via NJE). The RACF security token associated with the output is also made available.
When the SAPI application completes processing of a SYSOUT data set, it has a number of options on what to do with it (set in SSS2DISP). The primary options are to "process" the data set (same as a printer does when it prints a data set), keep the data set (with or without a system hold), JCL hold the data set, release the data set, and you can also ask that the data set not be shown to this address space or thread again. The do not show options apply until the instance (address space or thread) terminates or until some attribute of the output is changed ($T command). Additionally, there are a number of attributes about the output that can be altered..

SYSOUT API - Put/Get Advanced topics *(cont)*

- Do not have to completely process JOEs
  - Can change selection criteria mid JOE
  - Can change request type mid JOE
- One thread can process count, bulk modify, and PUT/GET requests
  - If selection or request changes, next call does a PUT with old rules and then new request
  - Acts like a "PUT/COUNT" or "PUT/BULK" request

When selecting JOEs for processing, you do not have to completely process the entire JOE you were handed. You can choose to process some data sets and then stop, or change what you are doing. For example, perhaps you want to process all output for a job once you select a job. You could do a selection based on a criteria like SYSOUT class, then when a JOE is selected, change your filters to the JOBID of the output you were handed and do a COUNT request (to see how much output there is for this job), then create a regular data set large enough to handle the output, and finally start doing PUT/GET calls with the JOBID filter. When all the available data sets for the job have been processed, you could update your filter to be just a SYSOUT filter again and look for another job. Note, when you switch requests (in this case change the type to count), you must define what to do with the data set you were handed (in this example, you would indicate to keep the data set). The count request in this case is really a "PUT/COUNT" request. It PUTs the JOE you were handed and then COUNT with the new filter.

SYSOUT API -
Put/Get Advanced topics *(cont)*

- All SYSOUT token types valid input for SAPI selection
  - JOE tokens - extended status
  - Client tokens - Allocation
  - Data set tokens - SAPI
- Terminate thread using SSS2CTRL
  - IAZSSS2 still validated
  - Check return code because request can fail
  - When in doubt, use SSS2PUGE type (least likely to fail)
  - Thread terminated when SSS2JEST is zero

Processing using SAPI can be enhanced by using SYSOUT token to select specific output to process. SYSOUT tokens take 3 forms. JOE tokens represent an entire JOE and can be obtained from the extended status interface. This allows the application to use extended status to determine which JOEs to process (using whatever selection process they want) and then using SAPI to actually process the data sets.
Client tokens are obtained at the time a dynamic allocation creates a SYSOUT data set. These tokens represent a single SPIN data set. Using client tokens, an application can directly allocate for processing a specific data set that was created earlier.
Data set tokens are part of the SYSOUT characteristics that are returned when SAPI selects a data set for processing. One way these tokens can be used is by selecting OUTPUT using normal SAPI selection but not actually processing the data set. The data set tokens can be saved and the data set returned with KEEP specified (and perhaps do not show to thread again). At a later time, data sets can be selected in the order that the application wants for processing. Or the data sets can be read using SPOOL browse in any order that the application wants and later deleted.
All SAPI requests create a thread. These threads must be terminated when no longer needed. Failing to terminate threads can increase system overhead. Threads are normally terminated when the creating task terminates. However, if multiple threads are being processed by one long running task, then the application must terminate the threads that are no longer needed by issuing a SAPI request with SSS2CTRL set. These calls go through normal validation processing but the only processing that occurs is the PUT portion of a PUT/GET. A thread exists if the creating task exists and SSS2JEST is non-zero. This is important because request with SSS2CTRL set can fail. When this happens, you may need to issue a request with SSS2PUGE set since it performs the least validation.

SYSOUT API -
Put/Get Advanced topics *(cont)*

- What about CLONE JOEs?
  - Multiple JOEs point to same set of PDDBs
  - Clone JOEs created when
    - /*JOBPARM COPIES=
    - $N PRTnnnn
  - Splitting PDDBs from one JOE would affect other (CLONE) JOE
  - Restriction: All data sets in a CLONE JOE MUST be process the same
  - SSS2DSH is on if processing CLONE JOE

Clone JOEs exist when multiple JOEs point to the same set of PDDBs. Third qualifier in JOE id indicate clone JOEs. JOE with ids of 1.1.1 and 1.1.2 are clones. Clone JOEs are created when JOBPARM COPIES= is specified (greater than 1) or a printer is repeated ($N PRTnnn command). They can also be created by an exit that does a $#ADD passing in an existing JOE as the prototype. The problem with clone JOEs is caused by the fact that PDDBs are associated with JOEs using the first 2 operands of the JOE id. So with clone JOEs, one PDDB is actually associated with multiple PDDBs. If SAPI were to move a PDDB to a new JOE, it would not only affect the JOE being processed, but also the clone JOE. This side effect can cause data to be lost. As a result of this, there is a restriction when processing clone JOEs that all data sets must be processed the same way (you cannot split data sets out of a clone JOE). SAPI sets the SSS2DSH bit when it is processing a clone JOE.

SYSOUT API - Program examples

- There are 3 sample TSO programs
  - All are authorized TSO commands
  - SAPICNT is sample count request
  - SAPIBULK is a sample bulk modify
  - SAPIOUT is a sample PUT/GET command
- Using a "?" option displays operand information

There are 3 example programs for SAPI. Each program explores a different type of the SAPI request. All are authorized TSO commands. SAPICNT and SAPIBULK are examples of count and bulk modify requests. The two examples are similar except that SAPIBULK includes modification operands that SAPICNT does not. SAPIOUT is a very primitive OUTPUT command that processes data sets that match the input criteria and displays the output on the terminal.
All samples take as input a "?" which displays simple help.

SPOOL Browse

- Not an SSI, rather parameter on dynamic allocation
  - Authorized allocation key (Browse token)
    - Mapped by IAZBTOKP
  - Can access any data set on SPOOL
    - SYSIN, SYSOUT, JCL
  - Data set can be busy on a device
  - Data set can still be open
    - Instorage (unwritten) buffers are available
      - On executing system only
  - Allocation can be by client or data set token

SPOOL browse is not an SSI, but rather a subsystem dynamic allocation. Processing is triggered by an authorized key that contains a browse token (mapped by IAZBTOKP). This browse token, along with the data set name, identifies the data set to browse. SPOOL browse can access any data set on SPOOL including SYSIN and SYSOUT data sets and the input JCL. Data sets can be busy on a device, or even still open by the creating address space. For data sets that are still open, the unwritten (instorage) buffers can be read (if the reading application is on the same system as the writing application). Browse can also use client or data set tokens to allocate a data set.

SPOOL Browse - IAZBTOKP

ID length (4)
ID ('BTOK')
Version length (2)
Type(0-3)
Version(3)
SPOOL token length(4)
SPOOL token
Job key length(4)
Job key
ASID length(2)
ASID
Receiver length(8)
Receiver
Log string length(0-255)
Log string length
Log string

There are 3 token types:

- 0 or 1 - Original browse token
  - SPOOL token is IOT MTTR
- 2 - Reserved for SAPI
- 3 - SYSOUT Token
  - SPOOL token is client or data set (SAPI) token

Receiver and Log string are for RACF checks

The browse token is used to identify to JES2 what data set to allocate. There are 2 forms available to applications (the third is reserved for use by SAPI). The first form sets a type field to either x'00' or x'01'. This form can be used by applications that do not have a data set or client token available. The other form that can be used by applications is a type 3 token (type field of x'03'). In this form, the application passes either the client or data set token of the data set to allocate in the IAZBTOKP.
Receiver and log string parameters are used for RACF calls.

SPOOL Browse - IAZBTOKP type 1

- Type 1 calls with SPOOL token = 0
  - DSN= is name of data set to allocate
    - can contain generic characters
    - DSN must have jobname and jobid
    - Job key is optional (will be validated if passed)
- Type 1 calls with SPOOL token <> 0
  - Token is MTTR of IOT with PDDB for DS
  - DSN= name of data set (no generics)
  - Job key is required

Type 1 calls come in 2 flavors. If SPOOL token=0 (BTOKIOTP=0) then JES2 will locate the data set to allocate from the data set name passed. The data set name is the standard JES2 SPOOL data set name
userid.jobname.jobid.Ddskey.dsname
In this flavor, the jobname and jobid must be specified. But the fields may be specified as generics. The minimum data set name is
.JOBNAME.J0123456.
If a job key is passed (BTOKJKEY), then it is used to ensure that the correct data set is allocated. However, the job key can be passed as zero.
The other flavor of type 1 calls specify an IOT spool address in the SPOOL token (BTOKIOTP = IOTMTTR). In this flavor, the full data set name must be specified with no generic, and the job key must be set. This is lower overhead than the first flavor but requires more knowledge of JES2 internals.

SPOOL Browse - IAZBTOKP type 3

- Type 3 calls pass a SYSOUT token
  - BTOKSPLT can be client or data set token
    - Client token from allocation
    - Data set (SAPI) token from SSS2DSTR
  - JOE token cannot be used
  - Job key (BTOKJKEY) is ignored
  - DSN= is also ignored

Type 3 calls pass in a SYSOUT token (client or data set) in BTOKSPLT. These are the tokens passed from a client token allocation or from a SAPI request (SSS2DSTR). JOE tokens from extended status are not supported. For type 3 calls, the job key and DSN= passed on the allocation are ignored.

SPOOL Browse - IAZBTOKP other fields

- ASID (BTOKASID)
  - If zero, no unwritten buffers obtained
  - If non-zero, attempt to get unwritten buffers
    - ASID of running address space that is writing SYSOUT
    - x'FFFF' have JES2 figure out ASID
- Receiver (BTOKRCID)
  - Passed to RACF as RECEIVER= on AUTH call
- Log string (BTOKLSDL & BTOKLSDA)
  - Passed to RACF as LOGSTR=

JES2 will attempt to obtain unwritten buffers if BTOKASID is non-zero. It can either specify the ASID on local system where data set is being written, or a x'FFFF'. A value of x'FFFF' tells JES2 to figure out what ASID the job is running on.
Receiver (BTOKRCID) is the userid the output is routed to (passed as RECEIVER= on RACF AUTH call). If the value passed is the same as the owner of the SYSOUT (in the token associated with the job) then RACF will permit access without needing a profile in the JESSPOOL class.
The log string (BTOKLSDL and BTOKLSDA) is information passed on the RACF AUTH call. This value is placed in the audit record for the request.

SPOOL Browse - Special DSN=

- Some nonstandard DSN= are supported
  - Type 1 calls with SPOOL token = 0

userid.jobname.jobid.JCL
userid.jobname.jobid.JESJCLIN
userid.jobname.jobid.JESJCL
userid.jobname.jobid.JESMSGLG
userid.jobname.jobid.JESYSMSG

  - JESMSGLG and JESYSMSG attempt to link spun JESLOG data sets

To allocate system data sets without having to know the data set key, you can use the special data set names specified. These data set names are only valid if the corresponding data sets exist. JCL and JESJCLIN refer to the input JCL for the job. JESJCL is the JCL listing (JCL images) that were output from the converter. JESMSGLG and JESYSMSG are the JES2 message log and system messages data sets. If the job specifies JESLOG=SPIN, the JESMSGLG and JESYSMSG data set names attempts to link together the SPIN data sets into one logical data set for processing.

SPOOL Browse - Allocation TUs

- Required allocation TUs
  - DALSSREQ - 'Subsystem request' to allocate a data set
  - DALBRTKN - Browse token
  - DALDSNAM - Name of the data set
  - DALSTATS - DISP = SHR
  - DALRTDDN - Return DDNAME - (optional)

When invoking allocation, the listed keys are needed. DALSSREQ is the name of the subsystem that the allocation is to be directed to.

SPOOL Browse - Program example

- BROWSE is sample application
  - Authorized program (not TSO application)
  - WTOR to request DSN and whether instorage buffers are wanted
  - Allocates data set
  - Uses IEBGENR to copy selected data set to output DD

The sample program uses WTORs to determine what data set to allocate and whether instorage buffers are wanted. It then allocates the data set (a type 1 token is used with SPOOL token=0). If the allocation succeeds, then IEBGENR is used to copy the SPOOL data set to an output DD.

Who-am-I SSI

- Gets subsystem information
  - SSI function 54 (IEFSSVI mapping macro)
  - Supports unauthorized callers
  - Directed SSI (does not require job structure)
- Returned information has fixed and variable section
- WHOAMI is program example
  - Program invokes SSI and issue WTOs

The Who Am I SSI (Subsystem Version Information) gets information about a subsystem. It supports unauthorized callers and directed SSI requests. Callers are not required to have a job structure associated with the JES2 that is processing the request. The output has a static section that is common to all subsystems. That is followed by a variable section with subsystem dependent data. The variable section has a KEYWORD='VALUE' format that is similar to REXX assignment statements.
The example program issues the SSI request and issues WTOs with the response.

Who-am-I SSI - Data returned

SSVIVERS=z/OS 1.5
SSVIFMID=HJE7708
SSVICNAM=JES2
NO USER DATA PRESENT,
JES_NODE='POK ',JES_MEMBERNAME='IBM1',
DYNAMIC_OUTPUT='YES',INITIATOR_RESTART='YES',
MULTIPLE_STCTSO='YES',FOUR_DIGIT_DEVNUMS='YES',
AUTO_RESTART_MANAGER='YES',
SAPI='YES',SAPI_CHARS='NO',
CLIENT_PRINT='YES',TSO_SYSOUT_CLASS='G,H',
WTR_SYSOUT_CLASS='A,B,C,D,E,F,I,J,K,L,M,N,O,
P,Q,R,S,T,U,V,W,X,Y,Z,0,1,2,3,4,5,6,7,8,9',
COMMAND_PREFIX='$'

This is the output from the sample WHOAMI SSI. The first 3 fields listed are in the base section of the SSOB (IEFSSVI). The later fields are the variable data that is returned.

Notify SSI

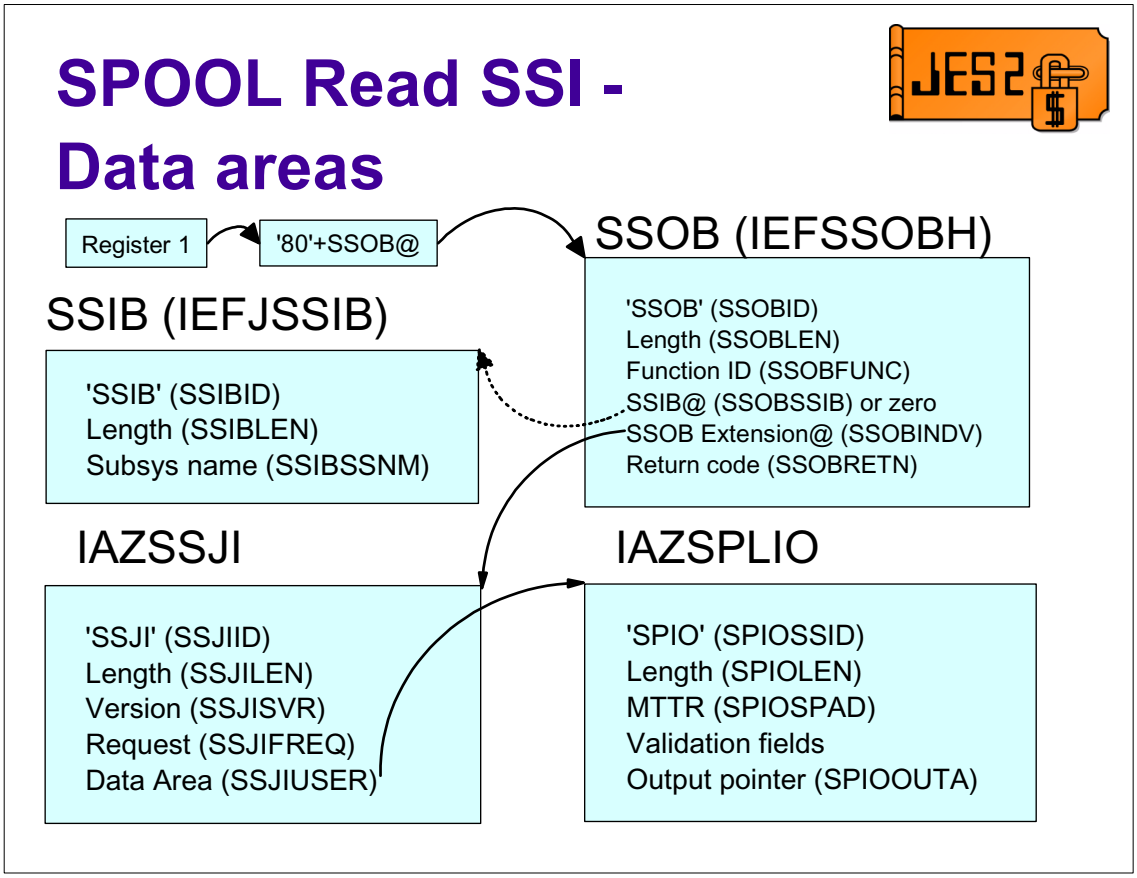
- Send notify message to user
  - SSI function 75 (IAZSSNU mapping macro)
  - Directed SSI (does not require job structure)
- Destination can be user on another node or member
- MSG is program example
  - Authorized TSO command to send a message to a node.userid
  - MSG ? gives quick help

The notify SSI can be used by an application to send a notify message to a user in the MAS or on another NJE node.
The sample program is an authorized TSO command that issued a message to a node.userid.

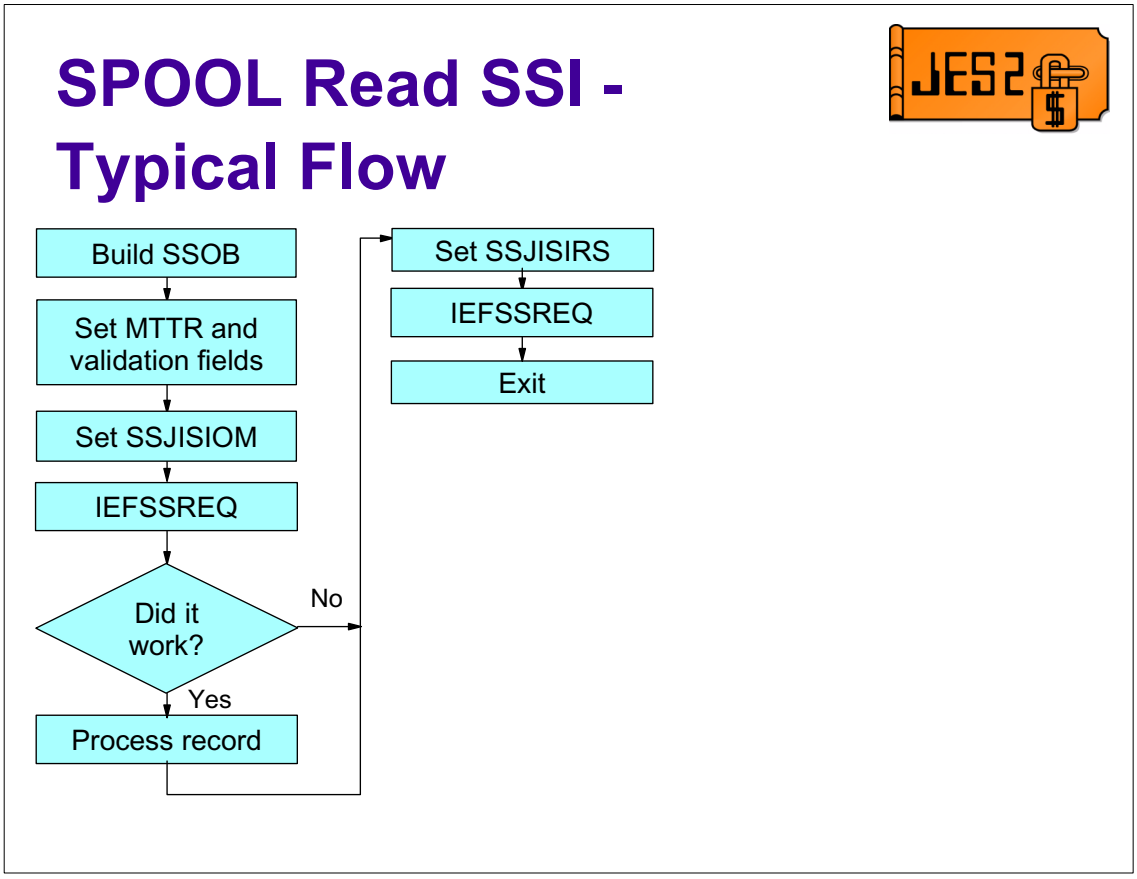
SPOOL Read SSI

- SSI reads blocks from SPOOL
  - Subfunction of SSI function 71 (IAZSSJI mapping macro)
  - Functions SSJISIRS and SSJISIOM (IAZSPLIO mapping macro)
  - Directed SSI (does not require job structure)
- Any MTTR can be read from SPOOL
- No security checks made

SPOOL read is a subfunction of the job information SSI 71. The SSI 71 SSOB extension is mapped by IAZSSJI. SSI 71 acts as a router for various JES SSI. The extension has a function code to identify what subfunction is needed and a pointer to a function dependent data area. The functions for SPOOL read are SSJISIOM (read data area) and SSJISIRS (free work areas).
SSI 71 requires callers to be authorized and supports directed SSI requests. Callers are not required to have a job structure associated with the JES2 that is processing the request.
This SSI will allow any record on SPOOL to be read including signature records associated with tracks. The SSI deals with relative vs. absolute track addressing.
There are no security checks made when using this SSI. Since the application using the SSI must be authorized, it is up to the application to ensure that the caller should have access to the data being read.



This is the data areas associated with the SPOOL read SSI. It is the normal data area associated with an SSI request plus the extra function dependent data area (IAZSPLIO).



Like other JES2 SSI calls, the SPOOL I/O SSI is a pair of calls. The first call reads in the data and returns a pointer to the data area. The second call returns the memory that was obtained. On the first call, the application must ensure that SPIOSTRP is set to zero. After that, the application must keep track of the IAZSPLIO, not altering SPIOSTRP, until the memory management call is made. If SPIOSTRP is non-zero, a memory cleanup call is needed. The memory associated with the SPOOL I/O request is owned by the task that made the first request and will be freed when the task terminates or when a memory management call is made. Multiple SSI requests can be made without a memory management call, however, the data from each call is read into the same output buffer (i.e. SPIOOUTA always will point to the same storage). If you want to have 2 data areas in storage at the same time, you must either copy the data to local storage or use one IAZSPLIO data area per request.

SPOOL Read - Program example

- SPOOLRD is an authorized program
  - Uses WTORs to determine what to read
    - MTTR, CB type, JOBNAME, JOBID, Job key, Data set key
    - Asks whether instorage buffers are needed
  - Output is WTOs about what was read
  - Print is dump format of record read

The SPOOLRD example is an authorized program that reads a single block from SPOOL. The input is obtained using WTORs. It includes not only the MTTR to read but also any validation fields that are to be passed. The ASID of a running address space can be passed to obtain unwritten (instorage) HDB data areas (only valid when a control block type of HDB is passed).
The output is a brief display via WTOs of what was read and a print data set that contains a hex dump of the record read.

Summary

- Gone through brief overview of newer SSIs
  - Some less new than others
- Examples are a good starting point for how to code to interfaces
- Real value comes in combining multiple SSIs
- More features are being added all the time
- Using the Subsystem Interface great starting point
  - Looking at DSECTs can fill in holes

The completes the quick tour of the newer SSI requests (and a couple older requests that may have been overlooked). The examples provided are intended as a starting point to build applications that interact with JES2. Though the examples tend to stress one interface at a time, the real value comes in combining interfaces to develop more complete applications.
These are not dead interfaces, they are constantly being updated with new features. New features are discussed at SHARE and in the Using the Subsystem Interface book. That book is a great resource for learning about the interfaces. However, looking at the DSECT that are involved in the interface often fills in some of the missing holes. SSI 71 is a great example. It is a router for many JES2 functions, not all are fully documented. Here is a complete list of the functions available and the related data areas:

```
SSJIFOBT (4)   IAZDSERV   CKPT version obtain
SSJIFREL (8)   IAZDSERV   CKPT version return
SSJIFJCO (12)  IAZJBCLD   JOBCLASS info obtain
SSJIFJCR (16)  IAZJBCLD   JOBCLASS info return
SSJISIOM (20)  IAZSPLIO   Read SPOOL block
SSJISIRS (24)  IAZSPLIO   Free SPOOL block storage
SSJICVDV (28)  IAZCVDEV   Convert device id
```

There are lots of things you can do with the SSI, all it takes is a little exploring

Questions?