Table of Contents

Notices....2
Introduction....2
Boot Loader Design....2
References....3
CPU Sharing....4
Subroutine CPU Sharing....6
The Boot Loader Environment....7
Boot Loaders....7
Boot Loader Services....9
BLSCALL – Calling a Boot Loader Service....11
BLSIOT – Boot Loader Services Input/Output Table Entry....13
BLSPB – Boot Loader Services Parameter Block....14
BLSTABLE – Boot Loader Service Table....15
SAVEAREA – The Boot Loader Save Area....16
System Related Services....18
Service ID – 0 – No Operation, NOOP....18
Service ID – 1 – Input/Output Initialization, IOINIT....19
Input/Output Related Services....21
Input/Output Table (IOT)....21
Operation Request Block (ORB)....25
Subchannel Status Word (SCSW)....26
Service ID – 2 – Query I/O Table, QIOT....28
Service ID – 3 – Enable Device, ENADEV....29
Service ID – 4 – Execute Channel Program, EXCP....32
Service ID – 5 – Pending Action Notification, PNDING....36
BLS – Calling Conventions – 32-bit Registers....37
Calling a Subroutine....39
Entering a Subroutine....39
Returning to the Caller....40
BLS – Calling Conventions – 64-bit Registers....41
Calling a Subroutine....42
Entering a Subroutine....43
Returning to the Caller....43
Appendix A – System Design Constraints....45
Appendix B – What APROB Does and Does Not Tell Us....48
Appendix C – EXCP Service Processing....50
Implications of Pending Actions....51
Appendix D – Boot Loader Structure and Service Implementation....53
The Device Targeted Boot Loader....53
Macros Internal to the Boot Loader....54
Boot Loader Services....54
Macros Internal to the Boot Loader Services....56

Notices

IBM and z/Architecture are registered trademarks of International Business Machines Corporation.

Introduction

Just loading a program is not the only capability that a boot loader can offer. It can provide some rudimentary services for itself and any booted program. In this role, the boot loader resembles a loadable Basic Input/Output System or BIOS.

This document describes how such services can be accessed in a standard way and documents any standard services offered by an SATK boot loader.

Boot Loader Services (BLS) are standardized subroutines called either by the boot loader itself or by a booted program. Subroutines are accessed by means of a BRANCH AND LINK (`BAL`) type instruction.

Boot Loader Design

A boot loader is assembled for a specific run-time architecture. The boot loader will validate that the run-time architecture and assembled architecture's "match". By "match" is meant that the run-time architecture is logically consistent for the assembled architecture. In most cases, these are identical. But in the case of z/Architecture® systems, the initial run-time architecture can be different. In this situation, the system will be placed into z/Architecture mode by the boot loader. In this case the boot loader has been assembled for z/Architecture. Other situations can be made to "match" by use of a PSW change.

Where the run-time and assembled architectures do not match, the boot loader will enter a disabled wait state.

Global system parameters are initialized by the boot loader. This includes the LOD1 record information supplied by the boot loader and the input/output system, in particular the enabling of all interruption sources. Input/output interruptions are controlled by the PSW.

Other than the IPL device, enabled automatically by the hardware, the boot loader or booted program must enable all devices used. The boot loader itself only uses the IPL device.

Additionally, the following design guidelines apply to boot loaders:

1. Boot loaders will reside below the 16-megabyte memory boundary.
2. Boot loader services will be **assembled** for 32-bit register modes, except for the z/Architecture case. The boot loader must be **assembled** for the z/Architecture case (ASMA `-t s390x` or `-t 64` options) to utilize 64-bit registers once it enters z/Architecture mode.
3. Boot loader services will utilize 24 (Format 0 CCW) or 31 (Format 1 CCW) -bit I/O commands depending upon the **assembled** operating environment's maximum capabilities.
4. Boot loaders will not utilize MIDAW's or IDAW's.

Correspondingly, there are implications for booted programs:

1. Booted programs will reside below the two-gigabyte memory boundary.
2. Booted programs are entered in the mode dictated by the boot loader.
3. Booted programs need not change the run-time architecture.
4. Use of MIDAW's or IDAW's and I/O areas above the two-gigabyte memory boundary can only be implemented by a booted program.

For a more thorough discussion of these guidelines refer to "Appendix A – Design Constraints".

References

Initial Program Load with ASMA, `doc/asma/iplasma.odt` or `doc/asma/iplasma.pdf`

Program Linkage, http://csc.columbusstate.edu/woolbright/LINKAGE.HTM

IBM® System/370 Extended Architecture Principles of Operation, SA22-7085-0, March, 1983.

Stand Alone Tool Kit Common Macros in `doc/macros/SATK.odt` or `doc/macros/SATK.pdf`

CPU Sharing

How a CPU is used will have implications for the operating environment. Early computer systems, until about the mid-1960's, were simple affairs (at least as computer systems are understood today). Those systems did not share the CPU. The computer system could execute a single program and that was it. They certainly had a CPU, a uni-processor. But were restricted to a single task. A "single tasking uni-processor" system. Reminds one of PC DOS on an early PC CPU. These provided, in essence, no sharing of the CPU. The single task essentially owned the memory. There were certainly explorations into more advanced usage that were able to inform the designs of the next generation of computing systems.

The next evolution mostly continued the use of a uni-processor but expanded the tasks to more than one. These were multitasking uni-processor systems. Systems such as MVT or DOS arrived. These shared the single CPU between multiple programs. The various programs shared memory too.

Two types of multitasking emerged. Cooperative multitasking allowed multiple "tasks" to share a CPU but they did so of their own free-will. Each task would give up control of the CPU to another task when **it** decided to do so. The tasks cooperated. A real-world example of this being CICS. In some respects CICS continues to be a cooperative multitasking environment even today. In this environment, a task only gives up control when it elects to do so. Usually the term "cooperative" implies working nicely with other tasks in the system. But it can be used to describe how the task interacts with the CPU itself.

The other type of multitasking is preemptive. The access to a CPU by a task is terminated by an outside event. The task is preempted from use of the CPU. The "outside event" is a CPU interruption. As the system complexity increased it became possible to not just share one processor, but share multiple processors. Any task can proceed on any CPU. This is a multitasking multiprocessor system.

Even single-tasking requires the sharing of the CPU between different portions of the single task. Such are usually referred to as subroutines. This is really about saving CPU state in memory allowing another portion of the same task to monopolize the CPU for a different purpose. Out of this need arose the concept of program linkage conventions. These are nothing more than a form of cooperation between different portions of a program perhaps written by different people.

A program does nothing in memory without the CPU. So memory sharing is really about how CPU's can share memory. The sharing of a single memory environment between multiple CPU's is really about sharing the state of the CPU to allow one or more tasks to use the CPU. Even in the early days of computing it was obvious that a system with multiple CPU's must be able to ensure memory is updated in a consistent way. On the System/360 systems an instruction, TEST AND SET, was used to lock and unlock memory. As the System/370 was being developed, a more robust mechanism was needed. This allowed for consistent setting of memory locations by means of the COMPARE AND SWAP

instruction. While this need in a multiprocessing environment was clear, it was not clear whether such a mechanism was required in a uni-processor system. Analysis demonstrated that in a preemptive multitasking environment, even with just one CPU, the instruction became necessary for consistent memory setting during critical memory updates. This drove the development of some forms of storage management using this instruction. An entire section within the *System/370 Principle of Operation* manual explained this need and demonstrated how to use COMPARE AND SWAP for such functions as enqueue/dequeue, wait/post and free-pool storage management.

Mainframes are inherently designed for **preemptive** multitasking. The act itself of preemption requires a mechanism for saving CPU state. The executing task ceases to execute instructions and a different task is initiated to at least capture the preempting condition. Preemption is implemented by the changing of the CPU's active Program Status Word (PSW). The inherently preemptive multitasking environment can be hobbled to only support cooperative multitasking or, at an even more constrained level, single-tasking. The preemptive multitasking environment can, through software, be made to appear as a single-tasking environment.

The "task" used to capture a preempting event is referred to as an interrupt service routine (ISR). In a single-tasking environment, the program itself becomes its own ISR. The program is only able to cooperate with the CPU for events that can be planned. Six types of preempting events exist in mainframe systems:

- machine-check conditions,
- external, for example, timing events,
- input/output,
- programming errors,
- system restart, and
- program requests in the form of supervisor calls.

With a single-tasking program, no resident operating system exists to which requests can be made by a supervisor call. Subroutines can be called but not an operating system. A system restart is completely random. It results from the action of an operator. Programming errors are never planned, although they do occur. Hardware error conditions too are random.

The single-tasking system can only plan for two types of events: some external related events, for example a timeout, and input/output events.

The hobbling of the other preempting events occurs through the use of a New PSW that puts the CPU into a disabled wait state for these events when encountered. The only mechanism to get out of that situation without a power-on is to manually issue a system restart. It too can be hobbled. SATK refers to the "hobbling" of these events by a disabled wait PSW as a "trap PSW".

Subroutine CPU Sharing

A subroutine is another form of CPU sharing. As with tasks, CPU state must be preserved across a subroutine call. The conventions around use of a subroutine were early developments for the mainframe. The conventions introduced in the 1960's persist to this day. The link in the references provides a good description of the conventions used.

The key to understanding the dynamics of these conventions resides in the nature of the save area used by the called subroutine when saving the CPU state of the calling routine. Typically these areas are static. The calling routine provides an area within itself for the save area. A statically allocated save areas allows for serial reuse. Serial reuse means that routine A can call subroutine B. Following the return by subroutine B, routine A can then call subroutine C, etc.

Static save areas ultimately limit the calling chain. In particular, recursion, a subroutine that ultimately calls itself multiple times is not possible. Limited recursion is allowed. For example. Let's assume routine A calls routine B. The save area used to store routine A's state (within routine A itself) is used by routine B. Assuming routine B has a save area within itself for calling a subroutine, B can call itself. However, because of the static nature of the save area, subroutine B can not call itself multiple times. Only once. Why? Because once the save area within subroutine B is used, it can not be used again until the subroutine returns and allows the save area to be reused. If multiple calls are attempted, the saved state of the first call by subroutine B is lost. A statically allocated save area can only be used for **one** subroutine call at a time by the save area's owner. That subroutine may be another subroutine or itself. Hence, very limited recursion.

Are atomic operations (for example COMPARE AND SWAP) necessary in a uni-processor single-tasking environment? The simple answer is no. By eliminating the possibility of executing any instruction sequences that can experience interruption ensures that all instruction sequences are atomic. It would be possible to add dynamically assigned save areas that would allow recursion (up to the limits imposed by the number of allocated save areas, of course). These could be assigned without concern for interruption. Hence, a free-storage pool of save areas assigned using COMPARE AND SWAP is not needed.

The Boot Loader Environment

A boot loader developed for the Stand Alone Tool Kit (SATK) operates within a very simple environment. The CPU and I/O architectures will dictate how things are done within each domain, but the boot loader's use of the CPU and memory is the same across all SATK boot loaders.

An SATK boot loader operates with a single CPU. That CPU is selected during the externally initiated IPL function and is the CPU to which the IPL function passes control of the system thereby initiating execution of the boot loader. This is usually the CPU with a zero address, but does not have to be. The external initiation of the IPL function will control how this occurs.

Once instruction execution by the boot loader starts, it operates with interruptions disabled. The only time it enables **an** interrupt is when it **expects** that interruption to occur. Primarily this is an I/O interruption. This could be a device that interacts with a system operator, a user interface, or a device used for data access. This is achieved by entering an enabled wait state. Instruction execution ceases due to the wait state. But the enabled interruption can "wake" up the CPU and instruction execution can continue because the CPU is enabled to allow the interruption. With this design, if the expected interruption does not occur, the boot loader will simply sit waiting for an interruption that never occurs.

The boot loader is in essence a uni-processor single-tasking environment. It can easily support traditional static save area's with limited recursion. Any services based upon this approach must be designed such that recursion is limited.

The passing of control to the booted program does not change the environment established by the boot loader. Clearly, a booted program could establish its own execution environment thereby abandoning that established by the boot loader.

Boot Loaders

ASMA has the ability to define a PSW or CCW structure based upon the assembler's target architecture. It uses an automatic `XMODE` (Execution Mode) setting driven by the assembler's `-t` command**-**line argument. While this ensures the correct format, it does not ensure the correct field values or run-time modification to a PSW or CCW. These cases may require inclusion of a macro or locally defined macro to ensure correct assembly.

Boot loaders use three PSW's. The entry PSW that starts loader execution that becomes part of the IPL function is static in nature. It can utilize the XMODE approach, use `PSW` rather than `PSWxxx` when assembling the PSW.

The PSW that causes the processor to wait for an I/O interruption requires different settings enabling I/O interrupts in the S/360 or S/370 BC-mode PSW (5-bits) than in all other cases (1 bit). The New I/O PSW used to resume CPU execution is also static. The first and last PSW's can utilize the XMODE

setting. The waiting PSW will require generation from within a macro that understands what each architecture mode needs for the waiting bits. That should not be an issue, because the actual execution of the I/O, which includes the waiting PSW, should probably occur from within a macro anyway as part of a service.

Handling of the I/O interruption code occurs within the I/O process. As with the waiting PSW generation, this should be generated based upon `&ARCHLVL`, the global arithmetic value set by the SATK `ARCHLVL` macro based upon the ASMA command-line.

Boot Loaders provide five basic functions:

- initialize I/O operations for reading a directed record (device and I/O architecture specific)
- reading a directed record using I/O operations (device I/O architecture specific)
- adjust for reading the next record (device and I/O architecture specific)
- moving the directed record's contents to memory (CPU architecture specific)
- transferring control to the booted program (CPU architecture specific).

Boot Loader Services

The path to finding a boot loader's service has as its foundation in the LOD1 record content. The field at +48 decimal (+30 hexadecimal), contains the address of the Boot Loader Service Entry. The boot loader itself must establish this address within the LOD1 record at address X'000270'. This field within LOD1 is only valid if the boot loader services flag byte at address X'000257' is **not** zero. The `iplasma.odt` or `iplasma.pdf` document does not define how this flag byte is used. That is left to the boot loader and is defined on this document.

Boot Loader Services use the save area linkage conventions described in later sections when accessing a service.

The boot loader services entry address in the LOD1 record is used for all boot loader service requests.

Each supported service is assigned an identifying number, a service ID. The flag byte used by an SATK boot loader is the start of invalid service ID's. It is the number of supported services plus 1.

All services utilize register 1 for the address of a Service Parameter Block (SPB). The minimum SPB is four bytes in length. The first two bytes contain the service ID of the request. The next two bytes may optionally be used by the service for request information. A service may also extend the SPB with additional request related information.

```
    Service ID      Service Info.              Service Parameter Block Extension (SPBE)
+0                +2                +4 . . .
```

In the case where one service calls another, general register must be preserved by the calling service. Otherwise, its content is lost when the second service is called. The easiest way to accomplish that is utilization of a different register when the service accesses the SPB(E).

An invalid service ID results in a return code of a negative integer. All other return codes are zero (successful) or a positive value indicating the nature of the encountered situation. Each service documents the failure or warning conditions.

The following table summarizes boot loader services.

Service ID	Name	Service Info.	SPBE	Service Description
0	`NOOP`	yes	none	Tests the service framework
1	`IOINIT`	yes	yes	I/O system initialization
2	`QIOT`	yes	yes	Query I/O table for a device
3	`ENADEV`	yes	yes	Enable a device for I/O use, added to the I/O Table
4	`EXCP`	yes	ORB	Perform an I/O operation with the enabled device
5	`PNDING`	no	yes	Locate pending I/O action processing
6	`SENSE`	yes	yes	Read sense data from the device (planned)

Care must be taken with assembler `USING` statements. If a service establishes addressing with a `USING`, it can "leak" out into other services which actually know nothing about it. Make sure that all established `USING` statements have a corresponding `DROP` statement at the end of the service, typically before the `SERVRTN` macro.

Equally important is the usage of R1 as the address of the SPB that is input to the a service called by another service. While it will be preserved across the service call, it will not be restored to the caller's input SPB address. It will necessarily point to the input SPB before calling the other service. Hence, upon return, R1 still points to the SPB of the service from which the program has returned. The original R1 contents used by the calling service is lost. When using the calling conventions described above, R1 must be considered as volatile and preserved either in storage or another register before one service calls another.

BLSCALL – Calling a Boot Loader Service

Source File: `lodrmac/BLSCALL.mac`

Macro Format:

```
[label]   BLSCALL  SPB=label,SA=label
```

The `BLSCALL` macro implements a call to a boot loader service. A Service Parameter Block plus extension, when required, and a save area must both be supplied. The service call is performed and a return code is supplied in general register 15.

Assembly Considerations:

The call is dependent upon the LOD1 assigned storage area for entry to the service. A base register must be assigned to the assigned storage area DSECT (not the LOD1 structure itself), usually general register 0.

The service ID is specified within the SPB. The creation of the SPB and extension when required is the responsibility of the program. No macros are supplied for this purpose. The layout of the SPB for all services can be accessed by using the `BLSPB` macro.

The service ID for each service, as an equated symbol, can be generated using the `BLSTABLE` macro. Because ID's can change, it is recommended that this macro and the generated symbols be used within the SPB for a requested service's ID.

Execution Considerations:

Examination of the return code is required. The return code will indicate a negative value if the requested service ID is not available. Testing of this condition can occur in all architectures using a `$LTR` operator synonym and branching as appropriate.

Valid return codes from a service can be inspected with appropriate transfer of control by means of a branch table. The branch table will usually immediately follow the preceding test, when present, or the `BLSCALL` macro itself. It will typically use general register 15 as an index and the sequence of branches will transfer control as appropriate. The following is an example of how the program may perform these tests.

```
BLSCALL ...
$LTR  15,15       Test for a bad service ID
$BN   IDBAD       If bad, handle the situation
B    *+4(15)      Test valid return codes
$B    OK          If 0, transfer to successful code
$B    WARNING     If 4, handle the warning/error condition
$B    ERROR       If 8, handle the error conditions
```

The test for a bad service ID is particularly useful during program development. It will typically result from incorrect addressing of the SPB. If not tested, the B instruction will transfer control to some unexpected location and typically result in a program interruption.

Note that the B instruction is used and not a $B when branching into the branch table. This is because access to the branch table requires an index register. $B can utilize instructions that do not have an index register in some architectures.

Label Field Usage:

The label field is associated with the first instruction generated by the macro when present.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
SPB	none	The label associated with the SPB of the service being called. If omitted the macro assumes that general register 1 has already been loaded with the address of the SPB.
SA	none	The label associated with the save area used for register preservation during the call. If omitted the macro assumes that general register 13 already contains the save area address.

Programming Note:

A save area is an area of 18 full or double words, depending upon the architecture of the CPU. The SAVEAREA macro may be used to create this area or the programmer can create it using the DC assembler operation.

BLSIOT – Boot Loader Services Input/Output Table Entry

Source File: `lodrmac/BLSIOT.mac`

Macro Format:

```
[label]  BLSIOT
```

The `BLSIOT` macro generates a DSECT allowing access to a boot loader Input/Output Table (IOT) entry. The boot loader table itself is considered a boot loader resource, but its entries are shared with service callers.

Assembly Considerations: None

Execution Considerations:

Depending upon the device involved, access to the IOT entry will vary from minimal to many uses. At a minimum, when a device is enabled by the `ENADEV` service, the caller must provide a device type that is derived from the information in the `BLSIOT` for that device. Most devices have values that should be used as documented by the `BLSTYP` field contents.

The `ENADEV` service returns the address of the entry created for the enabled device. Use of the device by a service caller will require use of the address provided by the `ENADEV` service.

The IOT, as created within the boot loader, is restricted to eight devices. The IPL device is always one of the eight devices. To support more entries, re-assemble the boot loader with additional devices.

The later section in this document, “Input/Output Table”, has more information about the table’s entries.

Label Field Usage:

The label field provides the name of the IOT entry DSECT. It is required.

Positional Parameters: None

Keyword Parameters: None

BLSPB – Boot Loader Services Parameter Block

Source File: `lodrmac/BLSPB.mac`

Macro Format:

```
BLSPB
```

The BLSPB macro generates a DSECT, SPB, that defines how each service uses the SPB.

Assembly Considerations:

DSECT SPB also contains coding examples for each service. These examples indicate how the SPB and SPBE, when used, can be coded. The DSECT is most useful when the program accesses the SPB or SPBE to supply information to the service or utilize the service results.

Execution Considerations: None

Label Field Usage:

The label field is not used by the BLSPB macro. Nor can the DSECT name for the SPB be changed.

Positional Parameters: None

Keyword Parameters: None

BLSTABLE – Boot Loader Service Table

Source File: `lodrmac/BLSTABLE.mac`

Macro Format:

```
BLSTABLE
```

`BLSTABLE` generates a sequence of EQUATE symbols corresponding to the boot loader service ID's.

Assembly Considerations:

> While use of the symbols is not required, changes to the boot loader services occur changing the ID. Use of the provided symbols will always generate the correct service ID.

Execution Considerations: None.

Label Field Usage: The label field is not supported.

Positional Parameters: None.

Keyword Parameters: None.

SAVEAREA – The Boot Loader Save Area

Source File: `lodrmac/SAVEAREA.mac`

Macro Format:

```
[label] SAVEAREA DSECT=[YES|NO|BOTH],REGSZ=[32|64]
```

SAVEAREA creates either a DSECT of a save area or an actual save area within the assembly. The size of the save area is determined by either the current assembly architecture level or, when specified, the register size, in bits, used by the CPU.

Assembly Considerations: None

Execution Considerations: None

Label Field Usage:

When a DSECT is not generated, the label field is associated with the start of the generated save area. When a DSECT is generated, the label field is ignored. A DSECT generated by this macro will have a fixed name along with its fields.

Execution Considerations:

The size of the save area must match that of the execution mode of the boot loader. In particular if the boot loader is using 32-bit registers, a booted program that might call the service must not change to 64-bit register usage independent of the boot loader.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
DSECT	NO	Three possible values area accepted: • YES – a DSECT for the usage of the register size is created • BOTH – two DSECT's are created, one for each register size • NO – A DSECT is not generated. Rather a save area in storage is assembled.
REGSZ	none	Specifies the size of the register used by the CPU: • 32 – Indicates that the CPU is using 32-bit registers. The save area or DSECT uses fullwords. • 64 – Indicates that the CPU is using 64-bit registers. The save area or DSECT uses doublewords. If omitted the register size is implied by the current assembly architecture level and is used for the generated DSECT or save area.

System Related Services

Service ID – 0 – No Operation, NOOP

The NOOP service is used by a caller to test its use of the service framework. It does nothing of significance, but validates that the caller is correctly calling a service and that the service framework is functional.

Programming Note: The NOOP service is useful during development.

Input Service Information: Signed value incremented by the service

Input SPBE: None

Output Service Information: Input value incremented by one

Return Code:

Return Code	Meaning
0	NOOP service successful

Service ID – 1 – Input/Output Initialization, IOINIT

`IOINIT` prepares the CPU for input/output operations. Part of this process is establishing the Input/Output table for use by the I/O system. The IPL device is initially added to the Input/Output Table of enabled devices.

It enables all possible input/output interruption sources for the given architecture. For channel based architectures, all channels are enabled. For channel-subsystem based architectures, all interruption subclasses are enabled. During I/O reset, a subchannel is assigned to interruption class 0. Once `IOINIT` has completed its functions, all input/output interruption masking is controlled by the PSW mask as provided by the output service information.

Programming Note:

The `IOINIT` service must not be called by a booted program. It is specific for use by the boot loader and is called only once.

Input Service Information:

Byte	Value	Description
0	See Description	IPL device class as indicated by the I/O Table • `X'00'` – Fixed-Block-Architecture DASD • `X'88'` – Count-Key-Data DASD • `X'90'` – Tape • `X'98'` – Card reader Card punch, printer, console and 3270 terminals are not eligible as IPL devices. For more information see the discussion in the `ENADEV` service.
1		Not used for input

Input SPBE: One fullword not used for input

Output Service Information:

Byte	Value	`ARCHLVL`	Description
0	X'01'	0-4	Only CCW Format 0 is supported by the Boot Loader services
	X'02'	5-9	Only CCW Format 1 is supported by the Boot Loader services
1			Not used for output

Output SPBE: One fullword containing the address of the IPL device's I/O Table entry when successful. Otherwise, not used.

Return Code	Byte 0	Byte 1	Byte 2	Byte 3
0	Address of the IPL device's I/O Table entry			
4	Not used			

Return Code:

Return Code	Meaning
0	`IOINIT` successful. Output SI and output SPBE valid.
4	`IOINIT` may not be called by the booted program

Input/Output Related Services

For about 18 years (1964-1982), the channel-based I/O was the standard. With the introduction of extended addressing (from 24-bits to 31-bits) in 1983, the channel subsystem was introduced. Device actions did not change. CCW's were still used. Status and sense data meant the same things. However, how the CPU communicated with the I/O system changed. The instructions changed for communicating the CCW's and informing the CPU of I/O status. This change in the CPU interface for input/output operations has been the norm ever since (37 years as of this writing). The channel subsystem interface remains in use even with the advent of direct 64-bit operations and TCW's.

From this historical perspective, the channel-based I/O system was the outlier. Initially, the channel subsystem, while causing the CPU interface to change, continued to use the traditional bit-parallel channel for device communication. Over time fiber based bit-serial communication became the norm. The channel subsystem itself gained functionality, but the basic forms of communication persisted. This system introduced some standard memory-based structures that persist: Operation Request Block (ORB), the Interruption Response Block (IRB), Subchannel Information Block (SCHIB), and some others.

Operating systems could hide these structures and related instruction usage from the programs using the operating system. For SATK, where no operating system exists, these structures can not be hidden. Where boot loader services are introduced, some of these structures can be subsumed into a service. In particular, the ORB is difficult to hide. It defines a direct interaction with a device, how to execute a channel program. The nature of the information supplied is very similar to that supplied by operating system SVC calls that do the same thing: execute a channel program (EXCP). This is the foundation of all device interactions. Boot Loader services focused on I/O operations standardizes on the ORB for initiating device communication. While S/360 and S/370 level systems lack an ORB, a boot loader service can still utilize it for the service interface. The goal is to make the service interface independent of the hardware I/O architecture so that a booted program can operate in any I/O architecture without change. Experience will demonstrate whether this is feasible.

Input/Output Table (IOT)

The Input/Output Table contains a structure used by the boot loader services in the management of input/output operations for a specific device. A device's IOT entry is created when the device is enabled. The IPL device is enabled by the hardware and does not need explicit enabling. The IPL device is added to the IOT by the `IOINIT` service used by the boot loader to initialize I/O operations.

The device entry in the table is 28 bytes in length.

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	Device Number (DEV)		Device Type (TYP)	Status
1	Hardware Device Address (HDW)			
2	Accumulated Status (UNCH)		Sense Data	Pending Action
3	Error Mask	Reserved		
4				
5	Subchannel Status Word			
6				
7	Reserved for booted program usage			

Device Number (DEV)

The device number is a static designation by which the program know the device. For Hercules, the device number is defined in the configuration.

Device Type (TYP)

The device type identifies the type of device by including descriptive information about the device and its interruption class. For a given set of assigned values, the only requirement is that they be unique.

For S/360 and S/370 the interruption class is purely informational. For later architectures the interruption class defines the device's interruption priority.

Two device attributes are identified in the Device Type: whether the device

- recognizes physical end-of-file conditions by signaling Unit Exception status, bit 0, or
- the device signals availability of data via an Attention interruption, bit 1.

In both cases, a value of 0 indicates the device does not have the attribute, and a value of 1 means the device does have the attribute.

Bits 2-4 specify the device's interruption class.

Bits 5-7 are available for additional device types. Note that all except the highest priority interruption class has been assigned. Any additional device types may share an interruption class with another device.

The following values are assigned. Bits 2-7 can be considered to be a device type independent of any attributes associated with the device.

PEOF Bit 0	ATTN Bit 1	Interruption Class Bits 2-4	Available Bits 5-7	Value	Device
0	0	000 - 0	000	X'00'	Highest priority available
0	0	001 - 0	000	X'08'	Fixed-Block-Architecture (FBA) DASD

PEOF Bit 0	ATTN Bit 1	Interruption Class Bits 2-4	Available Bits 5-7	Value	Device
1	0	001 - 1	001	X’89’	Count-Key-Data (CKD) DASD
1	0	010 - 2	000	X’90’	Sequential Tape Device (TAPE)
1	0	011 - 3	000	X’98’	Sequential input card reader (RDR)
0	0	100 - 4	000	X’20’	Sequential output card punch (PUN)
0	0	101 - 5	000	X’28’	Sequential output printer (PRN)
0	1	110 - 6	000	X’70’	Operator console (CON)
0	1	111 - 7	000	X’78’	Operator 3270 terminal

Status

The status of the device, its attributes, and information within the IOT entry are managed by this field.

Status Bit	Description
x...	When 1, device is busy. Turned on when device I/O operation successfully started.
.x..	When 1, the most recent SCSW is present in words 4-6.
..x.	When 1, an action is pending in word 2, byte 3 (Pending Action)
...x xx..	Reserved for future use.
.... ..x.	When 1, device and channel status present in bytes 2 and 3.
....x	When 1, sense data available in word 2, byte 2.

When the device busy status is set to one, bits 1, 2, 6, and 7 are all set to zero, essentially “clearing” all other status related information. The fields themselves are **NOT** cleared. Residual data from a previous operation may be present. The status bits should be interrogated before accessing the SCSW, pending action, device or channel, status or sense data fields.

Hardware Device Address (HDW)

The device address by which the hardware recognizes the device is placed in this field. This value is used to actually initiate I/O operations with the device.

For S/360 and S/370 the value in bytes 2 and 3 are identical to the device number and bytes 0 and 1 are ignored.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **Device Number**

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

For later architectures, the device address is specified by a Subsystem Identification Word (SIW). The SIW is a combination of a channel-set identifier (SSI) and subchannel number. The SIW is a four-byte value required by the hardware to be placed in general register 1’s low order 32 bits. It consists of two values and certain required bit position settings.

0 0 0 0 0 0 0 0 0 0 0 0 SSI 1 **Subchannel Number**

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
```

Initially the system enables 1 channel-set, channel-set identifier 00. Additional channel-sets can be enabled by means of an undocumented instruction, a CHANNEL SUBSYSTEM CALL.

Because the channel subsystem **requires** the hardware device address be placed in general register 1, SATK uses general register 1 for all I/O operations when addressing the hardware device.

Accumulated Status

Most I/O operations will require only one interruption. However, cases do arise where multiple I/O interruptions are required. Where multiple I/O interruptions can occur, the status from each interruption is combined in this field. When the I/O has presented all of the required status, the I/O operation is considered completed. This is usually channel end and device end unit status. The required ending status can be specified when the `EXCP` service is called.

When `EXCP` is called for the device, Status bit 6 is set to zero, indicating that any data in this field is invalid.

Sense Data

When Status bit 6 is one, this field contains the universal sense data byte. Sense data from a specific device can be up to 36 bytes. It is the responsibility of the program to read the sense data.

When a new channel program is initiated at this device, the Status bit is set to zero, indicating the sense data is no longer valid.

Pending Action

When Status bit 2 is one, this field contains an action that should be performed by the program. Currently two pending actions are recognized:

- X’00’ – No pending action for device,
- X’04’ – Program data is available for the program to read, and
- X’08’ – Sense data is available for the program to read.

Sense data can be made available by any device. Only devices that provide an Attention unit status interrupt will make data available to the program.

When a new channel program is initiated at this device, the Status bit 2 is set to zero, indicating the pending action has been satisfied, whether it really has or not.

The `PNDING` service notifies the program of a pending action condition.

Error Status

Used to detect a device error condition for the device. Created from the Device Type information provided when the device is added to the table.

Reserved

Reserved for future usage.

Subchannel Status Word

This field of three words (12 bytes) contains the most recent interrupt information supplied by the hardware. See the discussion below for the details of the format.

Program Reserved

This field is reserved for the booted program's use. It is never referenced or altered by the boot loader or its services. This allows the booted program to logically extend the information related to a device without physical changes to the IOT entry itself.

Operation Request Block (ORB)

The ORB in its generic form has the following structure:

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	Interruption Parameter			
1	Key	S 0 0 0 F P I A U 0 0 0	LPM	00000000
2	Channel Program Address			

The Interruption Parameter can be anything. SATK boot loaders use the Interruption Parameter for the I/O Table address of the device. The hardware address, required to actually initiate the I/O operations is accessed via the Interruption Parameter's content.

The Key field is the storage access key used for memory access by the channel or channel subsystem. For boot loader support, this field must be zero. It is valid for all architectures.

Bits S, P, I, A, and U are not used by SATK and must be zeros.

The F bit specifies the CCW format in use. For S/360 or S/370, this bit will always be 0. For later architectures it can be 0 (Format 0 CCW) or 1 (Format 1 CCW). SATK boot loaders will use Format 1 for later architectures. Otherwise it would not be able to load above the line.

Word 1, Byte 2, is the Logical Path Mask and must be zeros for S/360 or S/370 use. For later architectures this value may be used as described for 370-XA and beyond systems. Note, Hercules limits logical path access to just the first path, or X'80'. Hercules only supports one path.

Nevertheless, SATK boot loaders will use X'FF' for the LPM allowing any available path to be used in case a stand-alone program is used in other environments.

Word 1, Byte 3 must be zeros as described above.

Device number is a concept introduced with the channel subsystem. For S/360 and S/370, a device's number is the same as the device's channel and unit address. For later systems, the device number is assigned to a subchannel for convenience. It is a fixed designation for the device. Its subchannel number can change as the configuration changes. On a Hercules system, a device number is assigned in the Hercules configuration in the same way as is its channel and unit address.

The following tables identify the usage of the ORB. Fields in gray are ignored. Fields in green are required. Fields in blue are required in some cases.

For S/360 or S/370 architectures the ORB structure is this:

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	I/O Table entry address of the targeted device			
1	Key - 0	0 0 0 0 0 0 0 0 0 0 0 0	LPM - 255	00000000
2	Ignored	Channel Program Address		

For later architectures the ORB structure is this:

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	I/O Table entry address of the targeted device			
1	Key - 0	0 0 0 0 1 0 0 0 0 0 0 0	LPM - 255	00000000
2	Channel Program Address			

The only difference is the F bit which is 0 (CCW Format 0) for the early architectures and 1 for the later architectures. The Key is set to zero for use with the boot loader `EXCP` service, but can be other values.

Subchannel Status Word (SCSW)

When an interruption occurs, the interruption information is stored in a structure by the device or subchannel. The early architectures used an eight-byte structure for saving this information into the assigned storage area. When the channel subsystem was introduced the structure was increased to 12 bytes and contains additional information is a slightly different structure. The SCSW format is used for both architectures allowing the maximum information to be presented to the program.

In the following tables, gray fields are ignored. Green fields have meaningful information. Blue fields are channel subsystem specific.

For S/360 and S/370 architectures, the SCSW has this format.

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	Key	0 0 00 0		
1		CCW Address		
2	Unit Status	Channel Status	Byte Count	

For later architectures the SCSW has this format.

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	Key	0 0 00 1 0 0 0 0	Z E N 0 FC	AC SC
1	CCW Address			
2	Unit Status	Channel Status	Byte Count	

Service ID – 2 – Query I/O Table, QIOT

The `QIOT` service searches the I/O table for a device number, and if found returns the table entry's address.

Input Service Information: The device number being sought in the I/O table

Input SPBE: Ignored on input.

Output Service Information: The device number whose I/O Table address is returned. The input information unchanged.

Output SPBE: The supplied fullword containing the address of the I/O table entry when successful, 0 otherwise.

Return Code:

Return Code	Meaning
0	`QIOT` service successfully found the device number in the I/O table. Output SPBE contains the table entry's address.
4	`QIOT` service did **not** find the device number in the I/O table. Output SPBE set to zero.

Service ID – 3 – Enable Device, ENADEV

The ENADEV service enables a device for I/O operations with the program and adds it to the Input/Output Table of devices.

For channel attached devices, the device is validated as being operational and is then added to the I/O Table. Its device number and hardware addresses are identical in value.

For channel-subsystem attached devices when the device is available, it has

- its device number, as configured for its subchannel, and its subchannel identification word as its hardware address placed in the I/O Table.
- The subchannel enabled to the channel subsystem and its interruption class are based upon byte 0 of the SI.
- The interruption parameter of the SCHIB is set to the I/O Table entry's address for the device.

Programming Note:

The program must know the device number of all of the devices with which it plans to communicate.

The EXCP service requires the I/O Table address of the device executing the channel program. The program should preserve the returned address for later use. Alternatively, the program can use the QIOT service to determine a device's I/O Table entry address.

Input Service Information: The device number of the device being enabled (and added to the I/O Table).

Input SPBE: One fullword as described in the following table

Byte(s)	Value	Description
0	See Below	Device class to be placed in the I/O Table.
1-3	X'00'	Unused as input to the service call.

The device class has three components: two flags and a code indicating the type of device. Taken together the combination must be unique because it both identifies the device and its attributes.

The code serves double duty. It both identifies the type of device but it also, for the channel subsystem, is the I/O interruption code for the device. The code identifies which interruption sub class is used by the device. The code is also positioned so that it can be placed within the SCHIB without shifting.

The following table identifies how the device class is constructed and the resulting values for commonly occurring devices. The available bits can be used for additional categories of devices. The column labeled "PEOF" indicates the device recognizes physical end-of-file conditions by means of a

Unit Exception status. The “ATTN” column indicates whether the device presents unsolicited status to tell the program it has pending data.

Class	PEOF Bit 0	ATTN Bit 1	Code Bits 2-4	Available Bits 5-7	Device
X′00′	0	0	000	000	Highest Priority available
X′08′	0	0	001	000	Fixed-Block-Architecture (FBA) DASD
X′89′	1	0	001	001	Count-Key-Data (CKD) DASD
x′90′	1	0	010	000	Sequential Tape
X′98′	1	0	011	000	Sequential card reader (input)
X′20′	0	0	100	000	Sequential card punch (output)
X′28′	0	0	101	000	Sequential printer
X′70′	0	1	110	000	Operator console
X′78′	0	1	111	000	Operator 3270 terminal

The code is exclusively used for the interruption code. The EXCP service uses bits 0 and 1 for its processing. Adjustments and additions to the code and available bits are possible. The above is just an example of how the assignments can be made.

Output Service Information: The input information is unchanged.

Output SPBE: The supplied fullword contains information dependent upon the Return Code.

Return Code	Byte 0	Byte 1	Byte 2	Byte 3
0	Address of the enabled device’s I/O Table entry			
4	Address of the existing device’s I/O Table entry			
8	Input device class	Not used, ignore	Device Status *	Channel Status *
12	Input device class		Not used, ignore	
16	Input device class		Not used, ignore	
20	Input device class		Not used, ignore	

* When available

Return Codes:

Return Code	Meaning
0	**Success**: Device enabled and added to the I/O Table. Output SPBE contains the new table entry’s address.
4	**Warning**: Found device in the I/O Table. Output SPBE contains the existing device’s I/O Table entry address.
8	**Error**: Device is in an error state. Device not added to the I/O Table. Additional information may be available in the SPBE.
12	**Error**: Device is not operational, busy, or invalid. Device not added to the I/O Table.
16	**Error**: I/O Table full. Device not added to the I/O Table. Device state unknown.
20	**Error**: Device class of existing I/O Table entry does not match that of the device being added.

Service ID – 4 – Execute Channel Program, EXCP

The EXCP service does what the name suggests. It executes a channel program directed to a device whose information is resident within the I/O Table. This implies that the device has been previously enabled by the ENADEV service. The I/O Table entry address is placed in the ORB which is the SPBE for the EXCP service.

The service performs three steps:

1. Starts the requested I/O with the device.
2. Wait for an interruption(s) from the device and place the ending status in the I/O Table for the device. Interrupt waiting can require both channel end and device end status.
3. Analyze the results of the completed status from the I/O Table and establish the service return code.

Step 1 can be bypassed when indicated in the Input Service Information field. For devices which indicate when to read data by an attention interrupt, I/O with the device should bypass step 1 to wait for the interruption. Normally none of the steps are bypassed. The interruption wait process can be forced to wait for both channel end and device end. Byte 0 of the SI field controls these actions.

Operator Devices:

Two types of devices are supported by SATK that signal a need for the program to read user input data by an attention unit status: a console device, and a 3270 channel attached terminal. For either device, the interruption can be either expected (because the program called the EXCP service waiting for the interruption) or unexpected (because the program called the EXCP service for a different device). In either case, the EXCP service will return a code indicating the **a device** has data. Use the PNDING service to recognize this condition.

Recognition of the need to read from the device is only possible when the EXCP service is called. It may be from an interruption that is pending and recognized by the I/O system during the small window in which interruptions are briefly enabled, or when the service is called to just wait for such an interruption from an operator device. When the program is ready to process data from an operator device it should first use the PNDING service to determine if the device has already signaled that data needs to be transferred. If the return code from PNDING indicates the device has data, then the program should simply read the data (with EXCP) from the device. If there is no pending action required, the program should then wait for the interruption using EXCP and bypassing Step 1, that is, just wait for the interruption. The return code will indicate the device has data, and the program should proceed as it did in the previous situation, by reading the data.

Physical End-of-File Condition:

Some devices detect an “end of file” at the physical level. Card readers do this when the input hopper is empty. Tape devices do this when a tape mark is read. CKD DASD devices do this when a record is read of zero length. The program is informed of this condition by means of a Unit Exception status. EXCP will report this condition via a return code of 4. This should be considered a normal response to a read-type operation for such devices.

Sense Data:

Any device may have additional information to send to the program. This is indicated by a Unit Check status. When an interruption ends this status, a return code of 12 is sent to the program by `EXCP`. The program should respond by reading the sense data. Command Reject is of most interest to the program. This indicates something is wrong with the channel program. The issues are device specific and imply a programming error.

Programming Note:

To simplify the creation of the ORB, a simplified macro is provided, `BLSORB`. Internally it will use the `ORB` macro supplied by SATK, but takes care of most of the parameters for the programmer.

Input Service Information: Input SI data controls the operations performed by the service.

Byte	Step	SI Value	Description
0	1	`0xx. ....`	Do not bypass channel program start processing. Start the I/O.
		`1xx. ....`	Bypass channel program start processing. Do not start the I/O.
	2	`x0x. xx..`	Waiting for an interruption can not be bypassed.
		`x0x. 00..`	Do not wait for either channel end or device end status.
		`x0x. 01..`	Wait for only device end status
		`x0x. 10..`	Wait for only channel end status
		`x0x. 11..`	**Wait for both channel end and device end status. Note 1.**
	3	`xx0. ....`	Analysis of I/O status in the I/O Table can not be bypassed
1		ignored	Not used for input to `EXCP` service

Note 1: Most I/O operations will use an SI, byte 0, of X’0C’: Start the I/O, wait for the interruption(s) until both channel end and device end have been received without error.

Input SPBE: A three word ORB defining the I/O operations to be started at the targeted device. The following tables identify the usage of the ORB. Fields in gray are ignored. Fields in green are required. Fields in blue are required in some cases.

For S/360 or S/370 architectures the ORB structure is this:

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	I/O Table entry address of the targeted device			
1	Key - 0	0 0 0 0 0 0 0 0 0 0 0 0	LPM - 255	00000000
2	Ignored	Channel Program Address		

For later architectures, the ORB structure is this:

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	I/O Table entry address of the targeted device			
1	Key - 0	0 0 0 0 1 0 0 0 0 0 0 0	LPM - 255	00000000
2	Channel Program Address			

The only difference is the F bit which is 0 (CCW Format 0) for the early architectures and 1 for the later architectures. The Key is set to zero for use with the boot loader `EXCP` service.

The ORB must have an I/O Table entry address in all cases. The key, channel program address, and, for later architectures, the Logical Path Mask (LPM) are required when Step1 (start I/O operations) is **not** bypassed (bit 0 of the first SI byte is 0).

Output Service Information:

Byte	Value	Description
0	ignored	Not used. Input value unchanged.
1	See Description	Device(s) pending status: • X'00' – No devices with pending actions during this operation. • X'01' – At least one device with pending action requires program notification.

During processing of an I/O request, the CPU will enter an enabled wait state, waiting for an I/O interruption. When I/O interruptions are enabled, any I/O interruption source can present its interruption information. Devices will present such status when they have input to be transferred to the system or have sense data needing to be presented to the program. The `EXCP` service will be presented with the unsolicited status whether it wants it or not. The service will post such status into the I/O Table as a pending action. It is the program's responsibility to recognize a pending action and device by use of the `PNDING` service. The primary device may also signal a pending action.

Output SPBE: Input ORB unchanged. Bytes 0-3 still contain the address of the I/O Table entry being addressed.

Return Codes:

Return Code	Meaning
0	**Success**: Primary device completed the I/O functions without error.

Return Code	Meaning
4	**Warning**: Primary device detected physical-end-of file condition during the operation
8	**Error**: Primary device is in an error state. See the I/O Table entry for additional information.
12	**Error**: Primary device is not operational, busy, or invalid.
16	**Error**: ORB not valid for requested operation

Service ID – 5 – Pending Action Notification, PNDING

The PNDING service queries a specific device or scans the IOT in search of a device needing a program action. Only two actions are recognized by the PNDING service:

- program needs to read sense data from a device, and
- program needs to read data from a device that has pending data.

When a device has a pending action, the PNDING service's return code indicates the pending action and the output SPBE field identifies the device. PNDING does **NOT** clear the pending condition. Only use of EXCP with the identified device will clear the condition.

Program notification of pending actions is only required when the EXCP service indicates one or more actions are pending as a result of its latest operation. It is possible that servicing of a pending action (by calling EXCP) can result in additional pending actions. PNDING should be called until all pending actions have been notified to the program. This is indicated by a return code of zero.

Use of EXCP for a device that has a pending action, eliminates the pending action status regardless of whether the action is the one that is pending.

Input Service Information: Not used

Input SPBE: The IOT device entry being examined for a pending action or zero if the entire IOT is being inspected. If a specific device is being examined, PNDING will inspect only that device. The entire table is not examined. If the entire table is being examined, the inspection of device entries ceases when a pending action is found.

Output Service Information: Not used

Output SPBE: The IOT device entry for which the return code applies. May be zero if no devices need to notify the program of a pending action.

Return Code:

Return Code	Meaning
0	No pending action(s) for program notification. SPBE unchanged from input.
4	**Warning**: SPBE identifies the IOT entry of the device requiring data to be read.
8	**Error**: SPBE identifies the IOT entry of the device requiring sense data to be read.

BLS – Calling Conventions – 32-bit Registers

Calling a Boot Loader Service (BLS), centers around the usage of the save area for preserving CPU state during the call to the service. The save area is provided by the **caller** of the service. In the context of a boot loader, this can be either the boot loader itself during the loading process or the program loaded into memory by the boot loader.

The following table describes the standard save area as seen by the **calling** program. It is the responsibility of the **called** subroutine to ensure these are the contents of the calling program's save area. The total length of a save area is 72 bytes. The layout assumes 32-bit registers. The layout for 64-bit registers is identical but increases to 144 bytes.

Disp. (Dec)	Disp. (Hex)	State	Register	Description
0	0	undefined	none	Reserved for boot loader environment usage (originally used by PL/1).
4	4	static	none	Back pointer to calling program's save area, otherwise 0. Set by the called subroutine in its own save area to preserve the calling program's R13. See Description for R13, below.
8	8	static	none	Forward pointer to called subroutine's save area, otherwise 0. If used, must be set by the called subroutine. Only the called subroutine knows where its save area resides.
12	C	static	R14	Calling program's return address
16	10	volatile	R15	When a subroutine is called, this contains the address of the called subroutine's entry point. During the execution of the subroutine, the save area will contain the entry address of the subroutine called. Upon return from the called subroutine, the register contains the returned code to the calling program indicating the success or failure of the subroutine, by convention: • 0 – called subroutine successful • 4 – warning or failure condition occurred, or • 8 and above – failure condition occurred. Values are usually multiples of 4 expected to be used in a branch table. This value must be set in the register by the called subroutine **after** the calling program's registers are restored or altered within the save area before restoring the contents.
20	14	static	R0	Calling program's R0

Disp. (Dec)	Disp. (Hex)	State	Register	Description
24	18	static	R1	Calling program's R1 – Address of calling program's parameters passed to the called program
28	1C	static	R2	Calling program's R2
32	20	static	R3	Calling program's R3
36	24	static	R4	Calling program's R4
40	28	static	R5	Calling program's R5
44	2C	static	R6	Calling program's R6
48	30	static	R7	Calling program's R7
52	34	static	R8	Calling program's R8
56	38	static	R9	Calling program's R9
60	3C	static	R10	Calling program's R10
64	40	static	R11	Calling program's R11
68	44	static	R12	Calling program's R12
Not saved	Not saved	volatile	R13	Address of this save area passed to the called subroutine. The called subroutine must preserve this value during the call so that the registers can be restored from the save area before return to the caller. If the called subroutine does not issue any of its own calls, it can preserve R13 by simply not using the register. If the called subroutine calls its own subroutine, the contents of R13 will change. See the Description of the field at displacement 4.

The saves areas, as used above, create a chain of save areas. The following table illustrates the state of the backward and forward pointers depending upon the sequence of subroutines. For this diagram, program A calls subroutine B which in turn calls subroutine C. This sequence results in two save areas. The colors illustrate which component sets the pointer in which save area.

Pointer Usage	Program A	Subroutine B	Subroutine C
Back Pointer	zero	Program A's save area	Subroutine B's save area
Forward Pointer	Subroutine B's save area	Subroutine C's save area	zero

Subroutine C must ensure it can return to Subroutine B. It may use a save area for preservation of R13, illustrated above, or preserve it in some other manner. Correspondingly, the forward pointer in Subroutine B's save area may be zero if Subroutine C does not have a save area. So, everything in orange is optional.

The presence of forward pointer can be used to indicate whether the save area is available or not. If a forward pointer is present, then the save area is in use and recursion must be prohibited. This would also necessitate clearing of the forward pointer once a subroutine has returned.

While this section assumes that the called subroutine is supplied by the boot loader, there is nothing restricting a booted program from using the same mechanism when calling its own subroutines.

Calling a Subroutine

The calling program supplies the save area used during the subroutine call. R13 points to this save area before the call.

```
            LA    1,RTNAPRMS
            LA    13,SAVEAREA
```

The location of the called subroutine is placed in register 15.

```
            L     15,SUBRTNA
```

The call occurs by using a branch-and-link type instruction. For example,

```
            BALR  14,15     Other similar instructions may be used
RETURN      EQU   *
```

Control is passed to the called subroutine and register 14 contains the point within the calling program to which the called subroutine returns.

The values used during the call are coded this way.

```
SUBRNTA  DC    A(SUBRTN) Address of the called subroutine
SAVEAREA DC    18F'0'    My save area
RNTAPRMS DC    ...       Parameters passed to the subroutine
```

These are the three memory areas used to call a subroutine.

Entering a Subroutine

The subroutine must perform its responsibilities with regards to the caller's save area and its own, assuming it has one. The subroutine can use register 15 as its base register, or another. Register 15 is not a good choice if the subroutine calls other subroutines. The following example uses register 12.

```
SUBRTN   DS    0H           The subroutine entry point
         STM   14,12,12(13) Save regs in caller's save area
* It is now safe to use any register except 13
         LR    12,15        Use the entry point for addresses
         USING SUBRTN,12    This is needed for access to MYSAVE
         ST    13,MYSAVE+4  Save caller's reg 13
[        LA    0,MYSAVE     Update the caller's save area with
         ST    0,8(,13)     ..the address of my save area]
* Now 13 can be used safely as well.
```

At this point the subroutine can process its functions based upon the contents (and agreed upon meaning) of register 1. The last two instructions (the LOAD ADDRESS and STORE), in square brackets, are optional depending upon whether the forward pointer is being maintained or not.

Only one storage area is required for this logic, the subroutine's save area.

```
MYSAVE   DC    18F'0'
```

This save area is also used when the subroutine returns.

Returning to the Caller

Returning to the caller is as follows. It essentially does in reverse what was done upon entry.

```
         L     13,MYSAVE+4  Restore caller's register 13
* Update caller's register 15 with return code here (optional)
         LM    14,12,12(13) Restore caller's registers
* These are caller's register. Only alter reg 15 for return code
         DROP  12           Can not use reg. 12 anymore
         LA    15,4         Return a 4 as the return code
         BR    14           Return to the calling program
```

The subroutine has now returned control to the calling program with a return code. How the return code is set can be done in other ways.

Depending upon the needs of the subroutine, the caller's save area can be used for setting the return code. The called subroutine "reaches" into the caller's save area and updates the memory contents that will become the contents of register 15. In this case, the return code is set in the save area just after register 13 has been loaded with its address but prior to restoring all of the other registers. If the caller's save area is used for the return code, the setting of reg 15 just prior to returning to the caller is not needed. This is particularly needed if the return code has been placed in the subroutine's memory. Memory access requires a base register and so the updating of register 15 is impossible once the subroutine no longer has a base register. Note the location of the assembler `DROP` operation.

BLS – Calling Conventions – 64-bit Registers

The calling conventions for 64-bit registers are logically identical to 32-bit register usage, just twice as big.

The following table describes the standard save area as seen by the **calling** program. It is the responsibility of the **called** subroutine to ensure these are the contents of the calling program's save area. The total length of a save area is 144 bytes. The layout assumes 64-bit registers. See the discussion in the previous section for how the backward and forward pointers are used at displacements 8 and 16 respectively.

Disp. (Dec)	Disp. (Hex)	State	Register	Description
0	0	undefined	none	Reserved for boot loader environment usage (originally used by PL/1).
8	8	static	none	Back pointer to calling program's save area, otherwise 0. Set by the called subroutine in its own save area to preserve the calling program's R13. See Description for R13, below.
16	10	static	none	Forward pointer to called subroutine's save area, otherwise 0. If used, must be set by the called subroutine. Only the called subroutine knows where its save area resides.
24	18	static	R14	Calling program's return address
32	20	volatile	R15	When a subroutine is called, this contains the address of the called subroutine's entry point. During the execution of the subroutine, the save area will contain the entry address of the subroutine called. Upon return from the called subroutine, the register contains the returned code to the calling program indicating the success or failure of the subroutine, by convention: • 0 – called subroutine successful • 4 – warning or failure condition occurred, or • 8 and above – failure condition occurred. Values are usually multiples of 4 expected to be used in a branch table. This value must be set in the register by the called subroutine **after** the calling program's registers are restored or altered within the save area before restoring the contents.
40	28	static	R0	Calling program's R0

Disp. (Dec)	Disp. (Hex)	State	Register	Description
48	30	static	R1	Calling program's R1 – Address of calling program's parameters passed to the called program
56	38	static	R2	Calling program's R2
64	40	static	R3	Calling program's R3
72	48	static	R4	Calling program's R4
80	50	static	R5	Calling program's R5
88	58	static	R6	Calling program's R6
96	60	static	R7	Calling program's R7
104	68	static	R8	Calling program's R8
112	70	static	R9	Calling program's R9
120	78	static	R10	Calling program's R10
128	80	static	R11	Calling program's R11
136	88	static	R12	Calling program's R12
Not saved	Not saved	volatile	R13	Address of this save area passed to the called subroutine. The called subroutine must preserve this value during the call so that the registers can be restored from the save area before return to the caller. If the called subroutine does not issue any of its own calls, it can preserve R13 by simply not using the register. If the called subroutine calls its own subroutine, the contents of R13 will change. See the Description of the field at displacement 8.

The following sections are the same as for 32-bit registers, but updated with 64-bit register instructions where needed.

Calling a Subroutine

The calling program supplies the save area used during the subroutine call. R13 points to this save area before the call.

```
            LA      1,RTNAPRMS
            LA      13,SAVEAREA
```

The location of the called subroutine is placed in register 15.

```
            LG      15,SUBRTNA
```

The call occurs by using a branch-and-link type instruction. For example,

```
            BASR  14,15     Other similar instructions may be used
RETURN      EQU   *
```

Control is passed to the called subroutine and register 14 contains the point within the calling program to which the called subroutine returns.

The values used during the call are coded this way.

```
SUBRNTA  DC    AD(SUBRTN) Address of the called subroutine
SAVEAREA DC    18FD'0'    My save area (13 double words)
RNTAPRMS DC    ...        Parameters passed to the subroutine
```

These are the three memory areas used to call a subroutine.

Entering a Subroutine

The subroutine must perform its responsibilities with regards to the caller's save area and its own, assuming it has one. The subroutine can use register 15 as its base register, or another. Register 15 is not a good choice if the subroutine calls other subroutines. The following example uses register 12.

```
SUBRTN   DS    0H           The subroutine entry point
         STMG  14,12,24(13) Save regs in caller's save area
* It is now safe to use any register except 13
         LGR   12,15        Use the entry point for addresses
         USING SUBRTN,12    This is needed for access to MYSAVE
         STG   13,MYSAVE+8  Save caller's reg 13
[        LA    0,MYSAVE     Update the caller's save area with
         STG   0,16(,13)    ..the address of my save area]
* Now 13 can be used safely as well.
```

At this point the subroutine can process its functions based upon the contents (and agreed upon meaning) of register 1. The last two instructions, the LOAD ADDRESS and STORE (64), in the square brackets, are optional depending upon whether the forward pointer is being maintained or not.

Only one storage area is required for this logic, the subroutine's save area.

```
MYSAVE   DC    18FD'0'
```

This save area is also used when the subroutine returns.

Returning to the Caller

Returning to the caller is as follows. It essentially does in reverse what was done upon entry.

```
         LG    13,MYSAVE+8  Restore caller's register 13
* Update caller's register 15 with return code here (optional)
         LMG   14,12,24(13) Restore caller's registers
* These are caller's register. Only alter reg 15 for return code
         DROP  12           Can not use reg. 12 anymore
         LA    15,4         Return a 4 as the return code
         BR    14           Return to the calling program
```

The subroutine has now returned control to the calling program with a return code. How the return code is set can be done in other ways.

Depending upon the needs of the subroutine, the caller's save area can be used for setting the return code. The called subroutine "reaches" into the caller's save area and updates the memory contents that

will become the contents of register 15. In this case, the return code is set in the save area just after register 13 has been loaded with its address but prior to restoring all of the other registers. If the caller's save area is used for the return code, the setting of reg 15 just prior to returning to the caller is not needed. This is particularly the case when the return code has been placed in the subroutine's memory. Memory access requires a base register and so the updating of register 15 is impossible once the subroutine no longer has a base register. Note the location of the assembler `DROP` operation.

Appendix A – System Design Constraints

I/O has a single design constraint. I/O CCW residence must be within the first two gigabytes of memory. Period! Without TCW support, which allows for a TCW to reside above this limit, CCW placement is constrained. This is a mainframe design constraint. The I/O can be performed above this limit through use of MIDAW's or Format 2 IDAW's, but the CCW's **and** [M]IDAW's must reside below the limit. This applies to all bare-metal programs.

Booted program residency is dictated by how directed records are constructed. The directed record defines where in memory the record can be placed. It uses a four-byte field for the address. This means that booted program residency will reside within the first two-gigabytes of mainframe memory, in mainframe parlance, below the bar. This is consistent with the constraint on CCW residence. After all, the loading of a booted program by the loader is at its core an I/O operation.

The residency of the boot loader itself is further constrained by the mainframe IPL function. It only uses Format 0 CCW's. This means that any CCW's and data areas addressed by them must be below the first 16 megabytes of memory. No mechanism exists to allow a change to Format-1 CCW's during the IPL function or use of [M]IDAW's.

Format-0 CCW's can use an IDAW during the IPL function. But, the IDAW used by a Format-0 CCW is itself constrained to the first 16 megabytes of memory and must also reside in the first 16 megabytes because that is the design constraint of the Format-0 CCW itself. No functional advantage is provided by use of IDAW's with a Format-0 CCW during the IPL function.

As an aside, why do Format-1 IDAW's exist? Their purpose is to support virtual storage based I/O where the actual I/O area in absolute addressing terms is split between different physical pages of memory. The program residing in virtual memory sees the pages as contiguous, but the I/O system that only uses absolute addresses does not. The IDAW provides virtual to absolute address translation for the I/O system.

Nothing stops the booted program from relocating itself elsewhere, but, by **design**, that is where the program will reside, below the two gigabyte memory boundary. Can that design be changed? Sure. The directed record format could be changed and the boot loader could adapt to that change. The reality is that a form of CCW translation is required to perform the I/O operations addressing 64-bit locations. The MIDAW's or format-2 IDAW's must be generated at the time of execution. How to deal with 64-bit addressing while a program is being loaded and how such a booted program can perform I/O in a 64-bit world will need to be addressed. But why do that when all of the booted program's I/O CCW's must reside below the bar when executed.

While 64-bit enabled registers and instructions are relatively easy to enable, per se, how to design a boot loader such that it can readily adjust is another. At present, the anticipated approach is to assemble the boot loader for the environment it expects to utilize. This is why the LOD1 information around

expected environment should be tested by a boot loader and respond appropriately. How can the boot loader be designed to allow these differences without essentially creating three different loaders for each I/O device type (16 different loaders – four device types plus four operating environments) is still an open question. Adding services to the mix simply increases the need. Who wants to develop 16 loaders which essentially do the same things? Not me.

The other consideration for boot loaders is size. They really need to be very small. Ideally, a boot loader should be no more than 3,584 bytes in size. It can be stretched to 7,168 bytes. This drives the variation support into assembler macros via the macro global symbol `&ARCHLVL` and use of `$XX` instructions. The macro only provides its portion of the support for the targeted architecture.

Register size is another area of concern. For all systems that support 32-bit registers, common structures can be used. This includes all mainframe systems from the release of the first S/360 in 1964 to 2017. 64-bit registers were introduced in 2000 with z/Architecture systems while still supporting 32-bit registers until 2017 when the Configuration-z/Architecture-Architectural-Mode (CZAM) Facility was introduced. This facility creates a system that is in 64-bit mode immediately upon completion of the IPL function. New boot loaders for the CZAM environment will be required when this facility is introduced into Hercules. One lesson learned from the past is that 64-bit registers require different structures than does a system utilizing 32-bit registers. Technically of course both could be accommodated with structures designed for 64- and 32-bit registers. This typically increases the size of the structure, in some cases doubling it (for example register save areas).

Note, CZAM can be tested by changing into z/Architecture mode at the start of the boot loader. As far as the boot loader is concerned, it does not really care whether it changed into z/Architecture mode or was entered in it via a CZAM IPL function.

The IPL function, from the beginning to today, uses the Format-0 CCW. Regardless of the architecture supported by the system, any bare-metal program that participates in the IPL function (for example, a boot loader) will reside in the first 16-megabytes of memory. This is true even for systems that only support z/Architecture mode.

These musings drive these “keep it simple” conclusions:

5. Boot loaders will reside below the 16-megabyte memory boundary.
6. Boot loader services will be restricted to 32-bit register modes (except in the CZAM case).
7. Boot loader services will utilize 24 (Format 0 CCW) or 31 (Format 1 CCW) -bit I/O commands depending upon the operating environment’s requirements.
8. Boot loaders will not utilize MIDAW’s or IDAW’s.
9. Booted programs will reside below the two-gigabyte memory boundary.
10. Booted program entry will be in the mode dictated by the boot loader.

11. Booted program architecture changes are the responsibility of the **booted** program, not the boot loader (except in the CZAM case).
12. 64-bit I/O using MIDAW or IDAW is the responsibility of the **booted** program, not the boot loader.

Booted programs may utilize services offered by a boot loader or create its own. Interrupt service routines may be utilized for more advanced control of the environment. In this latter case the booted program abandons the boot loader facilities for its own.

Appendix B – What APROB Does and Does Not Tell Us

The APROB macro provided by the `maclib` directory of SATK, performs a "probe" of the CPU. It ultimately identifies the architecture level of the CPU. Aside from identifying the running architecture, it is also intended to provide a mechanism that ensures the architecture for which the program was assembled is compatible with the architecture executing in the CPU.

APROB can not do its job if there is not some level of compatibility. It requires an executing CPU, a CPU that is only executing instructions because the program's IPL PSW **is** compatible with the CPU architecture. For example, if the IPL PSW is in extended mode and has 31-bit addressing enabled, but the CPU is configured for System/370 operation, the program will never execute any instructions. The IPL PSW is invalid for this system. (System/370 does not support 31-bit addressing.) In this case, the APROB macro will never execute. In similar manner, if the IPL PSW is a BC-mode PSW and the CPU architecture is configured for ESA/390, the CPU will never execute any instructions. (ESA/390 does not support a PSW that is not in EC-mode.)

z/Architecture is a special case. As implemented by Hercules, the CPU will always start out in ESA/390-mode. That is architecture level 8. To get to z/Architecture-mode requires a SIGNAL PROCESSOR instruction. Once successfully executed, the current execution architecture level changes to 9.

As described above, there are certain cases that will guarantee an IPL failure. There are also cases that are, based upon the IPL PSW, ambiguous. For example, if the IPL PSW is for System/370 in EC-mode with 24-bit addressing mode, the CPU is unable to detect the difference from a 370-XA or above system also in 24-bit addressing mode. From the CPU alone, architecture levels 3-8 are essentially indistinguishable from the PSW alone when in 24-bit addressing mode. APROB **is** able to distinguish between these cases. For Hercules, APROB only needs to differentiate between architecture level 3 (System/370 in EC-mode) and 7 or 8 (ESA/390).

While all of these architectures are compatible to the extent that they run, they certainly have different instructions available. The most important being how the CPU accesses the input/output system. Even though APROB can differentiate these cases, the program must examine the results to determine whether a program assembled for a given architecture level is "compatible" with a given execution architecture.

The following table illustrates the situation for Hercules architectures. The top of the table identifies the architecture as assembled. The left side of the table identifies the APROB run-time results. At the intersection is the action that the program should take. Gray indicates that the assembled architectures are incompatible with the run-time architecture. Yellow indicates that the run-time architecture can be made compatible with the assembled architecture by using the instruction to change the run-time architecture. Yellow entries contain an instruction mnemonic indicates what is required to make the

two architectures compatible at run-time. Red means that the assembled architecture can not be made compatible with the run-time architecture during run-time but can be made compatible by changing the Hercules configured architecture. The blue entries are essentially red entries that may execute depending upon the IPL PSW contents. These too should change the Hercules configuration. Some cases are not possible when the program participates in the IPL process.

APROB	1	2	3	4	5	6	7	8	9
2	ok	ok	LPSW		ESA/390	ESA/390	ESA/390	z/Arch	z/Arch
3	LPSW	LPSW	ok		ESA/390	ESA/390	ESA/390	z/Arch	z/Arch
7	S/370	S/370	S/370		ok	ok	ok	z/Arch	z/Arch
8	S/370	S/370	S/370		ok	ok	ok	ok	SIGP
9	S/370	S/370	S/370		ESA/390	ESA/390	ESA/390	SIGP	ok

In the case of a booted program, some combinations are possible because the boot loader has already taken some action. This is true when APROB indicates the running architecture is 9. This is only possible on Hercules when the boot loader has transitioned to full z/Architecture.

There are definitely corner cases that can produce unexpected results. If the boot loader and booted programs are assembled for the same execution architecture (and Hercules is running with the proper configuration), the programs should run without issue. Mistakes do occur. This is an attempt to catch them and produce meaningful information for a resolution before the unpredictable occurs.

Appendix C – EXCP Service Processing

The EXCP service is driven by the program through the I/O Table (IOT) entry for a device. It primarily is intended for operations on a specific device, the primary device, identified in the SPBE ORB:

- Execute a channel program, or
- Wait for an Attention interrupt (interrupt with Attention set to 1 in the unit status)

The service must also deal with the situation that an interrupt can occur from some other device, the secondary device. For the secondary device, it only needs to store the interruption information in the IOT and indicate that further processing of the secondary device is required. Once this occurs, the service can return to working with the primary device.

A third situation exists for channel-based operations. An interruption is possible from a device that has not been enabled, unknown to the service. This is not possible for channel-subsystem operations. Only enabled devices, all of which are in the I/O Table, can actually cause an interruption with the channel-subsystem. Interruptions from unknown devices are ignored, as they would be with the channel subsystem.

The EXCP service is broken down into these steps:

1. Start an I/O operation with the primary device, if requested by SPB SI information
2. Wait for an interruption (store it in the IOT entry of the interrupting device)
3. Determine if interruption is from the primary, a secondary, or unknown device
4. Analyze the secondary or primary device interruption and record pending action.
5. If primary device and waiting for more status, return to step 2
6. If secondary device return to step 2. Still waiting for primary device interruption(s)
7. Unknown devices are ignored (and can only occur with a channel-based I/O system).

Step	Primary Device	Secondary Device	Unknown Device	Next Step
1 – start I/O	Yes, SPB SI	Not applicable	Not applicable	2
2 - wait		yes		3b
3a - interrupt			ignored	2
3b - interrupt	yes			4
4	next step 5	next Step 2		
5	need more status? - 2 Done			

The EXCP service interacts most directly with the I/O system of the underlying hardware than any of the other services. The service is developed and tested with use of the Hercules emulator. Certain realities of this environment control how successful the services will run in other environments, for example, a virtual machine. An emulator is not a simulator that creates an environment exactly like the physical environment. It should also be recognized that the S/360 and S/370 environment of channels, control units and devices does not exist even within the emulator. Hercules started out in the late 1990's as an emulator of the ESA/390 channel subsystem. The channel based environment was layered on top of the channel subsystem emulation. Most of the status conditions and responses that could occur in a physical channel environment simply do not exist in Hercules. Testing of these conditions are impossible with Hercules. While the coding of the services tries to maintain a perspective of the physical world that existed in that time frame, forgetting that it is running on Hercules, it is simply impossible to test thoroughly. Movement of these services to other environments where testing has not occurred will likely result in variable degrees of success. The author would be most interested in any reports of such.

Implications of Pending Actions

There is a small window into which an I/O interruption can occur. That window opens following the initiation of an I/O to a device and closes upon receipt of an interruption. By specifying that the I/O start be bypassed, EXCP goes directly to the I/O wait functionality, waiting for an interrupt from the primary device. The only way a secondary device can cause an interruption is for it to be already pending when the window opens or occurs sufficiently close in time that the interruption priority of the secondary device is higher than that of the primary device.

In either situation the pending action for the secondary device should be handled by the program sooner than the results of the primary device. That is of course impossible with the design of the EXCP service. The action being performed on the primary device will complete before the program even finds out about the secondary device's pending action (when `EXCP` returns with a pending action flag in the `SPBSI` field).

Both the primary and secondary device may require an action on the part of the program. The only actions that can be required are:

- reading sense data from a device, or
- reading data for the program from a device.

While the action requests can occur at the same time, the two actions can not occur for the same device. Additionally one is the primary device and one is the secondary device. The primary device request is always communicated by means of a return code from the `EXCP` service. The fact that a secondary device has a request for program action is communicated to the program by means of the `EXCP` service SPB service information field.

The only way to find out which secondary device is pending and what action is pending is to use the `PNDING` service. It will identify the device by its IOT entry address and the action that is pending via its return code. When should this occur? The service user program is the only one that can determine that. Most programs have a “processing loop”. Either at the beginning of the loop or at the end is a good place to position this. The nature of the pending actions may also dictate when to use `PNDING`. A read for sense data is logically providing information about the previous action. A read for operator data would logically be desirable at the beginning of the loop. However, the bottom of the loop can also be viewed as the top of the loop just at a different spot in the “loop”.

Appendix D – Boot Loader Structure and Service Implementation

Here is a general design of SATK boot loaders and the service structure. All macros are found in the `lodrmac` directory. All source files are found in the `lodrsrc` directory.

A boot loader is separated into two source files. The primary source used in the command line of ASMA that assembles the loader for a specific IPL device type, for example, `fbalodr.asm` performing the boot loader process from a FBA DASD volume. The services offered by the loader is included via an assembler COPY operation, ensuring each loader provides the same set of services. This source file is `bls.asm`. The services are required because the boot loader itself will use the services while loading the booted program.

Do to the nature of ASMA, macros are largely used due to the need for conditional assembly for the different target architectures. Much of the boot loader *source* is therefore identical between different targeted IPL devices. While half of the implementation for a targeted device is within the boot loader itself, the other half is in the `iplasma.py` tool. The tool actually builds the device content. The two are tightly coupled. Changes in one can force changes in the other.

This description of the boot loader technology is split into two sections. The first describes the macros used for conditional assembly, in the order they are encountered within the boot loader. The second describes the macros used by the services.

The Device Targeted Boot Loader

The following macros are used within the boot loader, found in the `lodrmac` directory. The `BLINIT` and `BOOTNTR` macros must not be used outside of the context of the boot loader.

Macro	Directory	Description
`ARCHLVL`	`maclib`	Establishes the assembly architecture level. Required for all other macros.
`BLASALD`	`lodrmac`	Establishes the PSW trap content of the first 512 bytes at assigned storage locations and the IPL PSW used by `iplasma.py`.
`BLINIT`	`lodrmac`	Initializes the boot loader. Save the Hercules IPL parameters. Establishes the base register. Loads the LOD1 with the boot loader's data. Determines the execution architecture. Adjusts if needed and possible to that architecture, including changing architecture mode.
none	none	Tests the boot loader services framework using the `NOOP` service. Initializes the Input/Output Table and adds the IPL device to the IOT using the `INIT` service. Ensures the IPL device can be found in the IOT using the `QIOT` service. Initializes the CCW's and related data for reading the booted program. Reads and loads the booted program's

Macro	Directory	Description
		directed records from the IPL device using the EXCP service.
BOOTNTR	lodrmac	Enters the booted program by setting its address mode and branching into the program based upon the LOD1 entry address.
	Loader Services - srcasm/bls.asm	
BLSPB	lodrmac	Service Parameter Block and extension usage by the services
BLSIOT	lodrmac	Structure of an IOT entry
SAVEAREA	lodrmac	Structure of a save area used with boot loader services
BLSIODS	lodrmac	I/O related structures used by the boot loader and its services
BLSASA	lodrmac	Hardware assigned storage areas and the LOD1 record definition

The line in the previous table that lacks a macro, indicated by the word “none”, is in essence the portion of the boot loader that is specific to the IPL device. Some macros exist to normalize access between the different CCW formats.

Macros Internal to the Boot Loader

Some of the macros are generally provided by SATK and some are specific to the boot loader. The Directory column indicates which is the case.

Macro	Used In	Directory	Description
ASALOAD	BLASALD	maclib	Creates the ASA trap PSW’s
ASAIPL	BLASALD	maclib	Creates the IPL PSW used by iplasma.py.
STLOD1	BLINIT	lodrmac	Stores into the LOD1 assigned storage area the content expected from the boot loader.
APROB	BLINIT	maclib	Perform a probing operation determining the execution architecture.
AARCH	BLINIT	lodrmac	Adjust to the execution architecture if needed.
ZARCH	BLINIT	maclib	Issue the instructions to convert to z/Architecture. Only generated when this is a possibility.
BLSCALL	Boot Loader	srcasm	Internal calls by the boot loader to its own services.

Boot Loader Services

Boot loader services are implemented within the lodrsrc/bls.asm source file. The following table identifies the macros used by this implementation. The boot loader services depend completely upon macros. Accommodating the different CPU and I/O architectures requires significant tailoring of the generated instructions.

Macro	Directory	Description
SERVS	lodrmac	Implements the boot loader services framework. The framework acts as an intermediary between the caller and the service. It locates a service’s entry address from its service ID.
BLSTABLE	lodrmac	This is the same macro as used by a program defining the service ID’s.

Macro	Directory	Description
		But when used within the boot loader itself, it also generates a table of entry addresses. Internally, the SERVID macro provides a service's entry address and its service ID, the same service ID exposed to other programs that call a service.
SERVNOOP	lodrmac	The NOOP service. Service ID: 0.
SERVIOIN	lodrmac	The IOINIT service used exclusively by the boot loader. Service ID: 1.
SERVQIOT	lodrmac	The QIOT service. Service ID: 2.
SERVENAD	lodrmac	The ENADEV service. Service ID: 3.
SERVEXCP	lodrmac	The EXCP service. Service ID: 4.
SERVPEND	lodrmac	The PNDING service. Service ID: 5.
BLCOMMON	lodrmac	Data shared between the various services and assembled as part of the boot loader.
BLMEM	lodrmac	Areas used by the boot loader but not assembled as part of the boot loader. Includes service save areas and the Input/Output Table itself. The DSECT generated by this macro, DMEMORY, must be made addressable during execution.

The addition of a service requires a number of steps.

1. Code the service in its own macro. It generates code as required by the assembly architecture level. See notes below.
2. Call the macro, generating the code for the assembly architecture level.
3. Any data shared between services must be added to BLCOMMON.
4. Any data that is strictly used during execution must be added to BLMEM, including the service's save area.
5. Add a SERVID macro for the added service to BLSTABLE.

When a new service is coded, it is initiated with a SERVICE macro. The service macro identifies the:

- service's entry address (referenced by the service's SERVID macro),
- service's required save area (added to BLMEM),
- service's service ID symbol (used for a storage eye catcher), and
- does some entry housekeeping for the service.

The service macro is terminated by the SERVRTN macro that returns to the services framework, which in turn will return to the service caller.

Macros Internal to the Boot Loader Services

The macros used internal to the boot loader services do two primary tasks:

- Contribute to the Boot Loader Service Table and
- Manage registers and save areas in conjunction with the services framework (SERVS macro).

Macro	Used In	Directory	Description
BLSERR	SERVIOIN SERVENAD	lodrmac	Common logic used in two services for creation of the I/O error device status mask based upon the device type attributes.
SERVEND	BLSTABLE	lodrmac	Terminates the Boot Loader Service Table.
SERVICE	each service	lodrmac	Initiates the service and provides information for SERVID.
SERVID	BLSTABLE	lodrmac	Creates an entry for a service within the Boot Loader Services Table.
SERVRTN	each service	lodrmac	Terminates the service and returns to the caller via the services framework.