

SATK User Guide

Table of Contents

Notices.....	1
Introduction.....	2
Building and Using the Sample Yourself.....	2
SATK (Bare-Metal) Programs.....	3
Recommended Documents.....	5
SATK Related Documents.....	5
Bitsavers Recommended Documents.....	6
Program 1 – Hello World.....	7
Step 1 – Assemble the Program.....	9
Step 2 – Run the Bare-Metal Program.....	10
Configuration File.....	10
Run Control File.....	11
List-Directed IPL Directory.....	11
Program 2 – Hello World From Disk.....	13
Step 1 – Assemble the Program.....	13
Step 2 – Create IPL Medium.....	13
Step 3 – Run the Bare-Metal Program.....	17
Program 3 – Hello World Using Console I/O.....	18
Assigned Storage Areas.....	18
CPU Initialization.....	20
Input/Output Initialization.....	21
Input/Output Operation.....	23
Input/Output Interruption.....	23
Input/Output Completion Analysis.....	25
Step 1 – Assemble the Program.....	26
Step 2 – Create the IPL Medium.....	26
FBA IPL Channel Program.....	28
Step 3 – Run the Bare-Metal Program.....	29

Copyright © 2020 Harold Grovesteen

See the file doc/fd1-1.3.txt for copying conditions.

Notices

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

“Python” is a registered trademark of the Python® Software Foundation.

SATK User Guide

Introduction

READ THIS INTRODUCTION!!!

Stand Alone Tool Kit, SATK, has a number of related features. SATK consists of a set of tools for the creation of bare-metal programs targeting the mainframe environment. While the most prominent might be its assembler, ASMA, a number of other tools are provided to assist in the bare-metal program's creation and use. It is targeted primarily for use with the Hercules mainframe hardware emulator, but is not restricted exclusively to it. Other emulators and virtual environments exist.

The tools provided by SATK are platform independent, exclusively using Python for their implementation. The author uses Linux to build, test and document SATK. As a consequence, this document, with its platform specific execution examples, is Linux oriented. My apologies to the Windows community. It is the author's hope, that a new Windows user can adjust. Contributions for use of SATK with Windows are welcomed.

This document is intended to give the new user of SATK help in effectively using it through example programs as well as where within the SATK document set additional information may be found. The programs as well as tool output is provided in the `samples/guide` SATK directory. Each program is numbered and is provided with its corresponding sample and output within `samples/guide/pgm x` , where x corresponds to the program's number in this document.

Hercules, the primary mainframe environment targeted by SATK, requires a configuration file to run. The configuration uses a filename suffix of `.conf`. After Hercules is invoked, a run control file, suffixed with `.rc`, can execute Hercules commands, as well as host platform commands via the Hercules `sh` command. These are placed in the `samples/guide/pgm x` directory. Each `pgm x` directory is self contained.

Assembler listings are too large to readily be placed in a document. To access the listing, a text editor will be required, such as Kwrite on Linux, or Wordpad of Windows. If available Visual Studio on Windows will also work. You never need to worry about line-terminations with SATK software. Python takes care of them. Other software may be sensitive to line-terminations.

Building and Using the Sample Yourself

It is recommended that you build the sample yourself in your own environment.

You will need the following files from the SATK sample program:

SATK User Guide

- `pgm \mathbf{x} .asm` – the assembler source of the sample program
- `pgm \mathbf{x} .conf` – the configuration file to run the sample with Hercules
- `pgm \mathbf{x} .rc` – the Hercules run control file that will launch the bare metal program

You will need to determine how you will execute `tools/asma.py`, `tools/iplasma.py`, if needed, and Hercules itself. Changing the working directory to the one containing the files from SATK will simplify your execution of the various tools. To facilitate this, additional Bash scripts are provided, one for each process:

- `asm` – Assembles the sample programs
- `med` – Creates the IPL medium (when used)
- `ipl` – Executes the sample bare-metal program in Hercules

It is also the goal to make migration to a different scripting language easier. These simple scripts should be easy to convert to a different scripting environment. Your directory will require a copy of `asm`, `med`, and `ipl` altered for your local environment and choice of scripting language. Once this is done for the first couple of programs, your local scripts can simply be copied and altered as needed.

Windows contributions are welcome.

SATK (Bare-Metal) Programs

All programs created by SATK are bare-metal programs. They execute without need of an operating system, interacting directly with the hardware environment, emulated or otherwise. As such there are at most three steps as described in this table.

Step	Required or Optional	Tool	Description
1	required	<code>asma.py</code>	Assemble a bare-metal program from source
2	optional	<code>iplasma.py</code>	Create IPL capable media from assembled program
3	required	<code>hercules</code>	Test the program by performing initial program load

`asma.py` and `iplasma.py` are provided by SATK in the `tools` directory.

This document assumes three things (from the previous table):

SATK User Guide

1. **Python®** is installed and running on the host platform. SATK requires **Python**. **Python** is available from <https://www.python.org/downloads/>. Generally a Windows installer is available from this site. On Linux, the operating system distributor usually has one in its package management system, if not already installed by the distributor.
2. SATK has been installed from github and ASMA runs. SATK is available at <https://github.com/s390guy/SATK>. The file `doc/asma/ASMA.pdf` provided with SATK describes getting started by verifying ASMA can be used. If ASMA, the assembler, runs, the other tools are also available. Building is not required.
3. The Hercules emulator has also been installed on your host platform. The latest version can be found at <https://github.com/SDL-Hercules-390/hyperion>. Pre-built Windows binaries can be found at <http://www.softdevlabs.com/hyperion.html#prebuilt>.

Instructions for building Hercules from source can be found in the github repository.

The development of bare-metal programs is an esoteric area of knowledge. This is true for mainframe systems. Any program that interacts directly with the hardware, that is, without an operating system, is a bare-metal program. An operating system, by this definition, is a bare-metal program. For the purposes of this document, “hardware” may be physical hardware or emulated hardware.

The distinguishing characteristic for bare-metal programs, at least in this document, is *how they are placed in memory*. To better describe this characteristic, three terms are used here to differentiate bare-metal programs:

- **IPL Program** – A bare-metal program *loaded into memory by the mainframe hardware* NOT requiring a boot loader program.
- **Boot Loader Program** – A bare-metal program that itself is loaded directly into memory as an IPL program by the hardware, but is *distinguished by providing the service of loading at least one other program from a boot medium into memory and transferring control to the loaded program*. A boot loader, by this definition, could provide other services, making them available to a loaded program.
- **Booted Program** – A bare-metal program that interacts directly with hardware but is *loaded into memory by a boot loader* from which control is passed.

Hercules supports a form of IPL program that does not require a typical external medium for the purpose of the IPL, for example a disk volume. This is called list-directed IPL. ASMA supports creating this as output from the assembler directly. Mainframes introduced this form of IPL when Linux started to be supported and installed from the host management console.

SATK User Guide

Hercules uses a directory for this purpose. ASMA can create this directory with the required content. This is why some samples do not require the use of `./run med`.

Key to the nature of bare-metal programs is from the above definition: *Any program that interacts directly with the hardware ... is a bare-metal program*. For mainframes, and most other systems, that means the bare-metal program operates in privilege program state. This is managed by the mainframe's PSW. The IPL PSW introduced to the CPU to start program execution must indicate privilege program state. When a bare-metal program starts execution it does so in real storage.

The bare-metal program must deal with the possible presence of interruptions, the most important being the program class of interruptions. Failing to do so will result in the CPU entering an interruption loop. The only way to get out of this is to manually stop the CPU. Stopping the CPU is done by use of the Hercules console command:

```
stopall
```

For the bare-metal program to “interact directly with the hardware” requires you, the programmer, to know exactly how to do that. Depending upon the architecture for which you are targeting your program, the appropriate *Principles of Operation* manual is strongly recommended. Most all of the details you will require are explained in this manual. Where external devices are involved, various other manuals will benefit for the specific device. For the earlier systems, there is a large amount of documents at www.bitsavers.org. These are largely S/360 and S/370 era documents, although some later documents are available. Much more recent documents can be located at the mainframe's manufacturer's websites.

Assigned storage areas used by the hardware and sometimes SATK must be honored.

Recommended Documents

The following documents are recommended. You already have those that are supplied by SATK. This document is one of them. Bitsavers, mentioned in the previous paragraph, has many of the documents you may find helpful. To locate other documents, and sometimes the ones on Bitsavers, your Internet search engine is your friend. The easiest way to potentially find a document is to search for its document number, particularly those from the mainframe's manufacturer. Your largest selection will likely be found by excluding the version number.

SATK Related Documents

SATK User Guide – `doc/guide.pdf` (this document)

SATK User Guide

ASMA – A Small Mainframe Assembler – [doc/asma/ASMA.pdf](#)

Stand Alone Tool Kit Common Macros – [doc/macros/SATK.pdf](#)

Initial Program Load with ASMA – [doc/asma/IPLASMA.pdf](#)

Undocumented Hercules – [doc/herc/Undocumented.pdf](#)

Hercules Fixed Block Architecture Emulation Reference Manual –
[doc/herc/FBA Manual.pdf](#)

Bitsavers Recommended Documents

Bitsavers occasionally reorganizes their site and links change. So the interested user will need to find the document.

IBM® System/370 Principles of Operation, GA22-7000-10, September 1987.

IBM System/370 Reference Summary, GX20-1850-2, November 1976.

SATK User Guide

Program 1 – Hello World

Program 1 is the legendary “Hello World” program. It does not require use of any I/O because it uses a DIAGNOSE instruction to deliver the message, part of Hercules VM emulation.

Program: samples/guide/pgm1/pgm1.asm

Hardware Architecture: S/370

Bare-Metal Program Type: IPL Program

IPL: list-directed

Macros: None

Program Description:

```
TITLE 'PGM1 - HELLO WORLD'
```

Gives the assembly listing a title.

```
PRINT DATA
```

See all object data in listing

See all of the data generated by the assembler.

```
PSWSECT  START 0,IPLPSW          Start the first region for the IPL PSW
* This results in IPLPSW.bin being created in the list directed IPL directory
PSWEC 0,0,0,0,PGMSECT The IPL PSW for this program
```

As with all bare-metal programs, it requires an IPL PSW to be placed in absolute addresses 0-7. To accomplish this a memory region is started, named IPLPSW, at address 0. The PSW reflects privilege mode and the address of the program itself, PGMSECT. In addition to starting a new memory region, it also starts a named control section. In the list-directed IPL control file, this area of memory becomes a binary image file named IPLPSW.bin. Both the second operand of the START operation (starting the memory region) and the PSWEC operation (generating a PSW in the same manner as a CCW) are ASMA specific.

```
PGMSECT  START X'300',IPLPGM1  Start a second region for the program itself
```

The IPL program is placed in a separate memory region, IPLPGM1. A second START operation is used for this. Unlike legacy assemblers, ASMA, accepts multiple START statements. The new control section, PGMSECT, has its own location counter set to absolute address X'300'. X'300' is selected to avoid hardware assigned storage areas and the SATK reserved areas in X'200'-X'2FF'. As with the initial START operation, the second one will create another file in the list-directed IPL directory, IPLPGM1.bin.

SATK User Guide

Both files created within the list-directed IPL directory are created from the control sections which make up each memory region. In addition, each file will be referenced, with its respective starting location, in the list-directed IPL control file within the same directory. The ASMA -g command line option specifies the path and file name of the control file. The directory for each of the binary files and control file is inferred from the -g argument. The three files are placed in the same directory.

```
BALR 12,0          Establish my base register
USING *,12         Tell the assembler
```

Usual program set up for its intended base register.

```
MVC X'68'(8,0),PGMEX Set a NEW PSW to trap program interruptions
PGMEX PSWEC 0,0,2,0,X'28' Points to the program OLD PSW
```

This is the earliest point in the program that a protection against an uninterrupted program interruption loop can be established. Why? Because the MVC instruction requires a base register. Once the base register is established, the program can place in the assigned storage area used by the hardware the program interruption New PSW. This PSW gets loaded in case a program interruption occurs. It stops the CPU automatically by having the wait bit set, the 2 in the third operand. All of the maskable interrupts are disabled. The combination of the two conditions effectively stops the CPU. Hercules will display such a PSW on the console, providing an opportunity to communicate why the program came to a halt. By using X'28' as the address, the PSW indicates that it was a program interruption and the Old PSW (where the condition occurred) is at real address X'28'.

```
LM      8,10,DIAGPRMS  Load the DIAGNOSE instruction's registers
DIAG    8,10,X'008'    Issue the Hello World message to the console
DIAGPRMS DC A(COMMAND),A(0),X'00',AL3(CMDLEN),A(0)
COMMAND DC C'MSG * '   VM emulated command
MESSAGE DC C'Hello Bare-Metal World!' Message text sent to self
CMDLEN  EQU *-COMMAND
```

This sequence initializes the registers used by the DIAGNOSE instruction, from DIAGPRMS. In turn, the DC's point to the message and contain its length, along with the flag byte.

```
BNZ BADCMD If a non-zero condition code set, the DIAGNOSE failed
LTR 10,10 Was a non-zero return code set by the DIAGNOSE?
BNZ BADCMD ..Yes, abnormal program termination
```

The results of the DIAGNOSE are checked here, first, the condition code and then the return code. If either fails the program branches to the label BADCMD. Instruction execution simply falls through if the DIAGNOSE was successful.

```
LPSW GOOD ..No, normal program termination
SPACE 1
BADCMD LPSW BAD End with a bad address in PSW field
GOOD PSWEC 0,0,2,0,0 Successful execution of the program
BAD PSWEC 0,0,2,0,X'DEAD' Unsuccessful execution of the program
```


SATK User Guide

Success or failure is indicated by which PSW is explicitly loaded by the program. In both cases a disabled wait state is established as was the case with the established Program New PSW. The difference is in the address field. For a successful program execution, an address of 0 is used. For an abnormal termination, an address of DEAD, all hexadecimal digits, is used. The LPSW instruction loads the PSW from memory into the active PSW.

So, the instruction address of the final program PSW indicates how the program was executed as in this table.

PSW Address	Meaning
X' 0000 '	Successful execution of the program
X' 0028 '	An unexpected program interruption occurred
X' DEAD '	An abnormal program termination occurred

Step 1 – Assemble the Program

Change the working directory to the directory into which you have moved the files from SATK.

Adjust the copied `asm` script with local environment changes (and adjust to your scripting language if needed).

The assembly is invoked with this command in the `asm` script:

```
{ASMA} -t s370 -d --stats -g ldip1/pgm1.txt -l asma-$sfx.txt pgm1.asm
```

The script variable `{ASMA}` points to the local location of the `asma.py` script within SATK's `tools` directory.

`asma.py` is presented with a number of command-line arguments. These affect the assembly in the following ways.

`-t s370` targets the S/370 architecture by the assembler. It restricts recognized instruction mnemonics to those recognized by systems implementing S/370.

`-d` influences the content of the listing. It causes ASMA to display in hexadecimal and characters the binary content it has created.

`--stats` causes `asma.py` to display on the console run-time time statistics and where time is spent. A statement rate is also provided.

SATK User Guide

`-g ldip1/pgm1.txt` identifies the type of output to be generated by the assembler, a list-directed IPL directory content, and where this content is to be placed. The `-d` argument prints the content of the binary files.

`-l asma-$sfx.txt` directs the assembler to generate a listing and into which file it is to be written. The use of the `$sfx` script variable allows the listing file name to include the date and time when it was created.

`pgm1.asm` identifies for the source file assembled by ASMA. It is a positional parameter and must always be the last in the command line.

Step 2 – Run the Bare-Metal Program

Change the working directory to the directory into which you have moved the files from SATK.

Execution of the program involves performing a list-directed IPL upon the `-g` file created by ASMA.

The IPL is initiated with these two commands in the `ipl` script:

```
export HERCULES_RC=pgm1.rc
${HERCULES} -v -f pgm1.conf >> log-${sfx}.txt
```

The first statement identifies to Hercules where to find the run control file, in this case `pgm1.rc`. This statement creates both a script variable and then provides it as a platform environment variable for access by Hercules. Script variables only exist within the script. Environment variables are made available to other programs.

The second statement actually launches the Hercules emulator. The `${HERCULES}` script variable identifies where on the local system the Hercules executable file is located.

The following command-line arguments influence Hercules execution.

`-v` causes Hyperion to provide verbose messages in the log.

`-f pgm1.conf` identifies to Hercules its configuration file.

The log file created by Hercules is sent to the file `log-${sfx}.txt`. As with the assembler listing, a date and time is added to the file name.

Configuration File

The Hercules configuration file has the following contents.

```
# Hercules sample configuration file for pgm1.asm
ARCHMODE S/370          # S/370 targeted by pgm1.asm
```

SATK User Guide

```
MAINSIZE 16K           # Sample starts at X'300'
NUMCPU 1               # Allow use of DIAG X'008'
```

```
# Console Device
000F 3215-C /         # Required by mainframe systems
```

ARCHMODE S/370 is specified because that is the architecture targeted by the program with the assembler's -t command-line argument.

MAINSIZE 16K is used because this is a small program and doesn't need much to run. Additionally, 16K will probably suffice for most bare-metal programs.

NUMCPU 1 specifies one CPU which is the norm for S/370 systems. And `pgm1.asm` does not require more than one CPU.

DIAG8CMD enable allows the DIAGNOSE function X'008' to be used by the program. In as much as this is how `pgm1.asm` conveys the Hello World message, it is required.

The single integrated console device is specified because mainframe systems require at least one console. It is available at address 000F.

Run Control File

The run control file is run after Hercules has configured the system with the configuration file. It contains Hercules console commands. All could be entered by means of the Hercules console itself. This simply automates the process.

The run control file has these contents:

```
# This file is intended for use with Hercules Hyperion
# Comment this statement if you do not want to trace
t+
# Perform the list-directed IPL
ipl ldip1/pgm1.txt
```

The `t+` Hercules console command causes instruction tracing to be performed. It is extremely useful during debugging of bare-metal programs, so is enabled here.

The `ipl` command performs the actual IPL, launching the bare-metal program from the list-directed IPL directory created by the ASMA -g argument.

List-Directed IPL Directory

The list-directed IPL directory's contents are created by the ASMA -g command-line argument. This directory contains three files related to the IPL function, all constructed by ASMA:

SATK User Guide

- `IPLPGM1.bin` – the bare-metal program's binary content as assembled (this is why the `ASMA -d` option is used).
- `IPLPSW.bin` – the IPL PSW used to start the execution of the program by Hercules following IPL.
- `pgm1.txt` – the IPL control file.

The IPL control file dictates where in memory the binary files are loaded, their starting memory addresses. The address was specified in the `ASMA START` operation's first parameter that initiated by binary content with its second parameter.

```
IPLPSW.bin 0x0
IPLPGM1.bin 0x300
```

When the `ipl` Hercules console command is executed by the run control file, this control file in the list-directed IPL directory controls where the file content is placed in memory.

The `IPLPSW.bin` file is placed at address `X'0'`. From the first 0 bytes of storage, Hercules initializes the PSW. After which, control of the CPU is passed to the loaded bare-metal program.

The Hercules console command `quit` may be entered to terminate Hercules execution.

SATK User Guide

Program 2 – Hello World From Disk

Program: `samples/guide/pgm1/pgm1.asm`

Hardware Architecture: S/370

Bare-Metal Program Type: IPL Program

IPL: FBA DASD volume

Macros: None

Program Description: same as Program 1

Program 2 is an extension of Program 1. It will use the output created by Program 1 to create a FBA volume that contains Program 1, the Hello World program. Copy from the `samples/guide/pgm2` directory into **your** Program 2 directory the following files:

- `med` – which builds the IPL FBA disk
- `ipl` – which IPL's the FBA disk.

Step 1 – Assemble the Program

This was already completed in Program 1. Program 2 uses the output from Step 1 in the previous program. Simply copy the contents of the `ldipl` directory in your Program 1 directory into the directory you are using for Program 2. Copy these files into the Program 2 `ldipl` directory. Do not change the file names!!

- `IPLPGM1.bin`
- `IPLPSW.bin`
- `pgm1.txt`

Step 2 – Create IPL Medium

Creation of the FBA IPL medium is accomplished with this command:

```
${IPLASMA} -v -f ld -m pgm2.3310 --records ldipl/pgm1.txt >> iplasma-${sfx}.txt
```

This command invokes `tools/iplasma.py` to build the FBA disk. As input to the tool, it uses the copied output from ASMA in Program 1, the list-directed IPL directory. This input file is the last argument of the command: `ldipl/pgm1.txt`.

SATK User Guide

Additionally the `-f ld` argument tells `iplasma.py` the format of the input file. `ld` indicates a list-directed IPL directory control file.

The output from the tool, the emulated IPL capable FBA disk file, is specified by the `-m` argument: `-m pgm2.3310`. The `-d` or `--dtype` argument defaults to `FBA` which implies a 3310 device type. Other FBA device types are supported.

The output from the tool is sent to a file in your Program 2 directory:

```
iplasma-mmddy-hhmmss.txt.
```

Two arguments influence the content of the output: `-v` and `--records`. `-v` requests verbose output. All lines lacking "iplasma.py" at the beginning of the output result because of the `-v` argument. When writing your own bare-metal programs, you will want this output.

The following explains the `iplasma.py` tool's output.

```
iplasma.py Copyright (C) 2015-2020 Harold Grovesteen
```

The usual copyright notice from the author.

```
iplasma.py - IPL HWM: 00036C
iplasma.py - Used HWM: 000370
iplasma.py - IPL PSW: 0008000000000300
```

The memory high water mark of the IPL and that actually used and the IPL PSW used by the IPL.

```
Volume:  TYPE=3310 SECTORS=125664 LFS=False
Host:     FILE=64339968 (61.3MB)
Block:    LENGTH=0 SECTORS=0 BLOCKS=None
```

Description of the selected device type and its present characteristics. These characteristics relate to the general device not necessarily the device constructed by `iplasma.py`. The general 3310 volume has 125,664 sectors. `LFS` indicates whether the host requires large file system support when accessing the FBA emulated file created with these characteristics. The `Host` line describes the full file size of the emulating file. The `Block` line indicates the file length, number of sectors and blocks currently created. These values should always be 0 without any blocks because these values are provided prior to the volumes actual creation.

```
FBA DASD Map:
IPL0: 0-0
VOLLBL: 1-1
IPLPGM1.bin: 2-2
```

SATK User Guide

How the FBA volume has been allocated for the IPL usage is described by these statements. The range indicates the beginning and ending sectors for each item. In this case, only one sector has been reserved for each major FBA disk item.

```
3310 IPL: LDIPL
```

This identifies the type of source installed on the FBA volume.

```
iplasma.py - IPL Record 1:
  000588 43000598 40000008 42000300 00000200
  000598 06000001 00000002
iplasma.py - IPL Record 0 - CCWs:
  000008 02000570 40000200 08000588 00000001
iplasma.py - IPL Record 0:
  000000 00080000 00000300 02000570 40000200
  000010 08000588 00000001
```

These statements identify where in memory (and the binary content for IPL records 0 and 1), in the order created by the tool. For IPL Record 0's creation, you must know where IPL Record 1 is placed in memory. Hence, IPL Record 1 is built before IPL Record 0. This is illustrated by the addresses in bold, X'000588'. The first address reflects where IPL Record 1 will be placed. This can then be used in IPL Record 0's creation of the Transfer-In-Channel CCW's address that causes the CCW program chain to continue with the CCW's in IPL Record 1. The following Memory Map is also required to know where these components reside.

```
Memory Map:
IPL0: 000000-000017
PSW: 000000-000007
CCW0: 000008-000017
IPLPGM1.bin: 000300-00036C
IPL1: 000588-00059F
```

Memory is allocated for the IPL process as described by these output lines. This is where you should find the specified content in memory. Addresses are in hexadecimal. Useful for debugging.

```
iplasma.py - emulated medium pgm2.3310 created with file size: 1536
```

The file created and its physical size in the host's file system is identified here. Note that the physical size is significantly smaller than that identified in the above `Host` output line for a generic FBA 3310 device.

This results from the `--size` argument defaulting to `mini`. Hercules supports non-standard device sizes. For such devices the device type of the configuration file dictates the characteristics of the device, not the file size. The underlying tools used by SATK for DASD creation allows for three size options: `mini` (the default used here), the absolute minimum file size required by the bare-metal program and supported by Hercules; `comp`, compression

SATK User Guide

eligible file; and `std`, the normal full size volume. SATK does not create compressed files. Either `comp` or `std` create uncompressed files that may be compressed by use of a Hercules utility.

```
FBA sector 0
 000000 00080000 00000300 02000570 40000200
 000010 08000588 00000001 43000598 40000008
 000020 42000300 00000200 06000001 00000002
 000030 00000000 00000000 00000000 00000000
...
 0001F0 00000000 00000000 00000000 00000000
FBA sector 2
 000000 05C0D207 0068C026 988AC03E 838A0008
 000010 4770C01C 12AA4770 C01C8200 C02E8200
 000020 C0360000 00000000 000A0000 00000028
 000030 000A0000 00000000 000A0000 0000DEAD
 000040 00000350 00000000 0000001D 00000000
 000050 D4E2C740 5C40C885 93939640 C2819985
 000060 60D485A3 819340E6 96999384 5A000000
 000070 00000000 00000000 00000000 00000000
...
 0001F0 00000000 00000000 00000000 00000000
```

These lines result from the `--records` command-line argument. The final output is that of each written sector on the FBA volume by `iplasma.py`. The redundant lines of binary zeros are indicated by the `...` in the above lines. The “address” on the left indicates the displacement into the sector where the data was placed. Sector 1 is not shown because the volume label has not been written to the emulated disk and hence is excluded from this text output. However, sector 1 is all zeros in the emulated disk.

For this disk, three sectors exist. 512 bytes per sector results in a file of 1536 bytes.

SATK User Guide

Step 3 – Run the Bare-Metal Program

The Hercules `pgm2.conf` file has added one line.

```
# Hercules sample configuration file for pgm2.asm
ARCHMODE S/370          # S/370 targeted by pgm2.asm
MAINSIZE 16K            # Sample starts at X'300'
NUMCPU 1
DIAG8CMD enable        # Allow use of DIAG X'008'

# Devices
000F 3215-C /          # Console required by mainframe systems
0110 3310 pgm2.3310    # FBA IPL disk
```

As can be seen, the FBA disk created by Step 2 has been added to the configuration file as device 110 (in hexadecimal). Otherwise it is identical to the configuration file used by Program 1.

Likewise the run control file has been changed to cause an IPL from the FBA disk created in Step 2.

```
# This file is intended for use with Hercules Hyperion
# Comment this statement if you do not want to trace
t+
# Perform the FBA based IPL
ipl 110
```

SATK User Guide

Program 3 – Hello World Using Console I/O

Program: `samples/guide/pgm1/pgm3.asm`

Hardware Architecture: S/370

Bare-Metal Program Type: IPL Program

IPL: FBA DASD volume

Macros: ARCHLVL, ASAIPL, ASALOAD, ASAREA, DWAIT, DWAITEND

References:

IBM System/370 Reference Summary

`doc/asma/IPLASMA.pdf`

`doc/macros/SATK.pdf`

Program Description:

Program 3 is another Hello World program, but what changes is how the output is delivered to the Hercules console. This program uses the console device to display the message using hardware input/output commands (as opposed to a DIAGNOSE instruction).

S/370 PoO, Chapter 13, Input/Output Operations, is recommended for details. “Control of Input/Output Devices” describes the various privileged instructions used by a bare-metal program. “Execution of Input/Output Operations” describes how the channel-command words are used to actually drive operations with a device. “Conclusion of Input/Output Operations” discusses how such operations end, the Input/Output Interruption, and the Channel-Status Word (CSW) which describes the conditions related to the I/O operation.

Assigned Storage Areas

By intent, this program uses few pre-defined storage structures. They tend to hide the basic actions of the code which is the objective of this program. However, DSECT's are valuable. The DSECT macro, a convenience for defining various DSECT definitions within a program. The format is either specified by the hardware or how SATK, software, uses the structure.

The hardware defines the usage of various areas of storage, generally referred to as Assigned Storage Areas. Most of these areas are within the first 512 bytes of storage, although additional areas exist in the second 4K-byte page. Two macros are provided by

SATK User Guide

SATK for definition of these areas: ASAREA and ASAZAREA, respectively. The ASAZAREA macro applies only to z/Architecture® assigned areas and is not discussed here.

```
ASA          ASAREA DSECT=YES
```

The ASAREA macro defines the areas assigned by hardware between X'0' and X'1FF' regardless of architecture. In general, the area between X'200'-X'11FF' is available for program usage. In the context of hardware specified memory usage, SATK is the “program.” SATK defines how the storage from X'200'-X'11FF' is used. The following table identifies each real memory area used by SATK. EOM stands for “end of memory”. The red areas are assigned by hardware. The blue area is required when a Boot Loader is in use. The gray area is reserved by SATK for future usage. All of the white areas are optional as needed by the program.

Area	Start	End	Length	Description
A	000	1FF	512	Assigned by hardware
B	200	23F	64	Hercules IPL command parameters
C	240	28F	80	LOD1 IPL Record 4
D	290	2FF	112	Reserved by SATK
E	300	11FF	3840	Small IPL program or Boot Loader
F	1200	1FFF	3584	Assigned by hardware
G	2000	EOM	varies	Any IPL program or Booted program

The bare-metal program will interact with any of these areas. How each area or field is used by the assigner should be respected by the program.

```
ARCHLVL
```

Architecture level is an SATK concept. Do you care about this? Not in this program. It is specific to S/370 using Extended Control mode. So why is it there? Many SATK macros are architecture sensitive. They generate different things based upon the architecture level of the program. The following two macros are sensitive to the architecture level of the program.

An MNOTE message is always produced by ARCHLVL informing the user which architecture has been identified. In this case the architecture level is 3. 3 stands for S/370. So, the correct target (ASMA command-line argument -t) has been specified when the assembler was initiated by the program.

```
ASASECT  ASALOAD
          ASAIPL IA=PGMSTART      Define the bare-metal program's IPL PSW
```

SATK User Guide

These two macros fill in the ASA with interrupt trap PSW's (ASALOAD) and an IPL PSW (ASAIPL). ASALOAD creates the ASA region image placed within the list-directed IPL directory, naming it the default ASAREGN. Note that both the region and control sections created by ASALOAD are symbols within the assembler's symbol dictionary. They must be unique.

CPU Initialization

PGMSECT START X'300',IPLPGM3 Start a second region for the program itself
The IPL Program itself has its own memory region and control section, IPLPGM3, and PGMSECT, respectively. It's load address is X'300'

USING ASA,0 Give me instruction access to the ASA CSECT
Because the program uses the ASA fields as defined by the DSECT created with the ASAREA macro, a USING operation is required to establish its addressability. Because it is within the first 4K bytes of memory, base displacement addressing does not require an active base register. Register 0 (ignored by such instructions) acts as a "filler". The assembler does not know the difference.

PGMSTART BALR 12,0 Establish my base register
USING *,12 Tell the assembler

The usual program initialization of its (real) base register.

MVC RSTNPSW,PGMRS

The ASA area for a console operator restart of the program is at real address X'0'-X'7'. Prior to the IPL these same addresses are assigned to the IPL PSW. I/O commands can only access absolute storage. However, to support multiprocessing, the CPU can map the first 4K, or 8K when using z/Architecture, bytes of storage to different absolute locations by means of the prefix register. Real storage, as seen by the CPU can be different than the same absolute addresses. Real storage then is a mechanism exclusive to the CPU. I/O operations know nothing about it.

The system reset that occurs at the start of the IPL function, will set the prefix register to 0. So, for a bare-metal program immediate following IPL, the locations X'0'-X'7' refer to the identical physical storage addresses for both modes of memory access: absolute (the I/O of the IPL) and real (instruction execution) storage.

So why does the bare-metal program care? Maybe it doesn't. But Program 3 is not designed to be reentered by an operator initiated restart. It is not serially reusable. And these 8 bytes, as real addresses, still contain the IPL PSW (as I/O data from the IPL). Left alone, the

SATK User Guide

operator could initiate a restart using (now real addresses) the IPL PSW, entering the program as if it had just been IPL'd.

To ensure this does not happen, the Restart New PSW, at real addresses X'0'-X'7' are initialized after IPL with a trap PSW.

Input/Output Initialization

```
LCTL  2,2,CR2      Enable only channel 0 for I/O interruptions
CR2    DC    XL4'80000000'  Enable only channel 0 where console is connected
```

When the system reset occurs at the start of the IPL function, control register 2 is initialized to allow interruptions to be presented to the CPU, provided the PSW allows them, for each channel attached to the CPU. This would be channel 0, for the console device (00F), and channel 1, for the FBA IPL disk device (110). In other words, CR2 would contain X'C0000000'. By loading control register 2, used for controlling interruptions to the CPU from various channels, with X'80000000', the FBA disk can not interrupt the CPU even if an interruption were to occur. If it were to occur, the program would fail with an error condition. By loading control register 2 with this value, makes the program a bit more fail proof. This takes care of the channels.

```
LH      1,CONDEV      Set up I/O device address in I/O register
TIO      0(1)          Determine if the console is there
CONDEV   DC    XL2'000F'  Console device address
```

Now, the subchannel, well, really, the channel, subchannel (control unit) and device are inspected to determine if they are operational, the complete path is functional or not.

The TIO (TEST IO) instruction sends a special command to the device, one not possible via a CCW. This command is X'00', a value invalid for the command field of a CCW. The command is treated as an immediate command (no data transferred) by the control unit. How the command succeeds or fails will indicate the state of the console device's path and will, accordingly, influence the return code set in the PSW by the TIO instruction.

The TIO instruction uses the storage format. The first (and only) operand identifies the channel/unit address of the targeted device. The storage format of the instruction, typically allows two ways that this address is supplied:

- as a hard coded address, for example, TIO X'00F' (using just the displacement field of the instruction), or
- as a generic value placed in a register, for example TIO 0(1).

SATK User Guide

Program 3 uses the register approach. Before executing the `TIO` instruction, general register 1 is loaded with the channel and unit address from a field initialized at assembly time, `CONDEV`.

The `TIO` instruction attempts to send the command, restricted for its use, to the device identified in the instruction's operand. `X'00F'` in this case.

```
BC      B'0100',DEVNOAVL  ..No, CC=1 might have a different address
BC      B'0010',DEVBUSY   ..No, CC=2 console device or channel is busy
BC      B'0001',DEVCSW    ..No, CC=3 CSW stored in ASA at X'40'
```

Depending upon the condition code setting, different error cases are identified. If a branch is taken it passes control to a LPSW that establishes a terminating error condition.

If none of the three branches is taken, then the console device is available on the path specified by the device address. The final step in initializing the input/output operation can proceed.

```
                MVC      CAW(8),CCWADDR      Identify in ASA where first CCW resides
CCWADDR DC      A(CONCCW) Address of first CCW executed by console device.
CONCCW  CCW0     X'09',MESSAGE,0,MSSGLEN      Write Hello World message with CR
*          CCW0     X'03',0,0,1                ..then a NOP.
* If the preceding NOP CCW command is enabled, then the CONCCW must set
* command chaining in the flag byte, setting the third operand to X'40'
MESSAGE DC      C'Hello Bare-Metal World!'    Data sent to console device
MSSGLEN EQU     *-MESSAGE                     Length of Hello World text data
```

A field within the ASA is reserved to identify where the CCW chain resides for execution by the channel, the channel address word (CAW). In this case, an address field is moved to the ASA, `CCWADDR`. In turn, this field points to the location of the CCW chain at `CONCCW`.

The CCW chain consists of a single CCW, a Write with CR (carriage return), command code `X'09'`. The CCW points to the data being sent to the console, namely, the Hello World message at `MESSAGE`, and its length, `MSSGLEN`. The only instruction here moves an address to the CAW. All the other assembler statements provide the initialization for the input/output operation itself.

Why is a NOP CCW, command `X'03'`, commented out? On good authority, while unlikely when working with Hercules, in the real world it is possible for the ending status to be delayed or missed. By chaining the NOP CCW to the Write with CR command, ending status is sure to be presented to the application.

Everything is now ready to output the Hello World message.

SATK User Guide

Input/Output Operation

Everything is now ready to actually send the Hello World message to the console. This is accomplished by the `SIO` (Start I/O) instruction.

```
SIO    0(1)           Request console channel program to start, did it?
BC     B'0100',DEVNOAVL ..No, CC=1 don't know why, but tell someone.
BC     B'0010',DEVBUSY  ..No, CC=2 console device or channel is busy
BC     B'0001',DEVCSW   ..No, CC=3 CSW stored in ASA at X'40'
```

The `SIO` is similar in design to the `TIO` previously used. It requires the device address, still in general register 1 from the `TIO`. It too sets the condition code based upon the results of the operation at the device when the CCW command is sent to the device. Unlike the `TIO` that sends a specific command to the device, `SIO` sends the command as specified in the first CCW. Following the reception of the command by the device, an initial status is sent to the channel attached to the CPU. As with the immediate command sent by the `TIO` instruction, the `SIO` will set the condition code based upon the device's response to the command.

The branch instructions detect anything that represents a failure and as with the `TIO`, transfers control to one of these locations in the program. They each load their respective PSW's and abnormally terminate the program with the designated instruction address.

```
DEVNOAVL LPSW NODEV      Code 004 End console device is not available
DEVBUSY  LPSW BUSYDEV    Code 008 End because device is busy (no wait)
DEVCSW   LPSW CSWSTR     Code 00C End because CSW stored in ASA
NODEV    DWAIT PGM=03,CMP=0,CODE=004 Console device not available
BUSYDEV  DWAIT PGM=03,CMP=0,CODE=008 Console device busy
CSWSTR   DWAIT PGM=03,CMP=0,CODE=00C CSW stored in ASA
```

By the process of elimination, success is recognized and the CPU continues with the next sequential instruction. At the device, the command has been accepted and the transfer of the Hello World message to the device is occurring. Hence this comment in the program.

```
* Console device is now receiving the message (CC=0)
```

Input/Output Interruption

The transfer to the device of the message continues until it is done some time in the future. What is the program to do during this time? Wait for the transfer to complete. Some discussion of program interruptions in general is useful at this point.

The mainframe has six interrupt classes. Most provide some information about the interruption's cause. Some can be masked off or on. If an interruption class is masked off and an interruption of that class occurs, it will remain pending until such time as the class is masked on. Other interruptions are unmasked and are always enabled. The following table

SATK User Guide

summarizes the general aspects of each class. See Chapter 6 in the S/370 PoO for details. Addresses are real within the ASA.

Priority	Class	Class Mask	Interrupt Mask	Old PSW	New PSW	Interrupt Information
0	Machine Check - Exigent	None	None	X'30'	X'70'	None
1	Supervisor Call	None	None	X'20'	X'60'	X'88'-X'8B'
2	Program	None	None	X'28'	X'68'	X'8C'-X'8F'
3	Machine Check - Repressible	PSW	CR 14	X'30'	X'70'	X'E8'-X'EF'
4	External	PSW	CR 0	X'18'	X'58'	X'84'-X'87'
5	Input/Output	PSW	CR 2	X'38'	X'78'	X'BA'-X'BB', CSW X'40'-X'47'
6	Restart	None	None	X'08'	X'00'	None

Regardless of the class, an interruption stores the active PSW and any related interruption information into the class' ASA old PSW area and its interruption information. This is followed by loading the active PSW from the class' ASA new PSW area. Before an interruption occurs or can occur due to masking, the class' New PSW must be set with meaningful PSW content. If this has not occurred, results are technically unpredictable, but usually result in an interrupt loop. Very unpleasant! The trap PSW's provide a valid PSW and will disable all maskable interruption classes if loaded by an unexpected interruption.

As seen previously, CR 2 was initialized to allow only input/output interruptions from channel 0. However, the I/O new PSW currently has only the trap PSW. The active PSW was loaded by the IPL function and it has input/output interruptions disabled.

The next instruction addresses the issue concerning the I/O new PSW.

```

DOWAIT   MVC   IONPSW(8),CONT   Set PSW for after I/O interrupt
          LPSW  WAIT           Wait for I/O interruption and CSW from channel
IODONE    EQU   *              The bare-metal program continues here after I/O
          MVC   IONPSW(8),IOTRAP   Restore I/O trap PSW
CONT      PSWEC 0,0,0,0,IODONE   Causes the CPU to continue after waiting
WAIT      PSWEC 2,0,2,0,0       Causes CPU to wait for I/O interruption

```

It copies a different PSW, at label `CONT`, into the I/O New PSW ASA. This PSW, like the IPL PSW, disables maskable interruptions, including any input/output interruptions, and does not cause the CPU to wait. In other words, when loaded by an interruption, the CPU begins executing instructions at the location `IODONE`.

The instruction following the `MVC`, the `LPSW` actually changes the active PSW to become a waiting PSW. The CPU ceases executing instructions. However, in addition to waiting, this active PSW has enabled I/O interruptions. These two instructions and two PSW's work in

SATK User Guide

conjunction to cause the wait for the I/O interrupt and continue CPU instructions after the interruption has occurred. The very next thing that occurs is restoring the trap PSW for I/O interruptions in case another one occurs.

Input/Output Completion Analysis

Now that the program knows the I/O is completed to the device, how well did it go? First, because there is no control over which device might present an interruption when enabled, was it the console that provided the interruption?

```
CH      1,IOICODE      Is the interrupt from the console?
BNE     DEVUNKN        ..No, end program with an error
DEVUNKN LPSW NOTCON    Code 010 End unexpected device caused I/O interruption
NOTCON  DWAIT PGM=03,CMP=0,CODE=010  interruption from some other device
```

General register 1 still has the console device address, so it is used to examine the I/O interruption code, the device address that triggered the interruption. If it is not the console device the program loads a disabled wait PSW to indicate a device unrecognized by the program caused an interruption.

Now that it is known that the interruption came from the console, the I/O operation's status can be checked in the Channel Status Word (CSW).

```
OC      STATUS,CSW+4    Accumulate Device and Channel status
STATUS  DC  XL2'0000'   Used to accumulate unit and channel status
```

One of the realities that the program must address is that the ending status of the channel and the device are usually presented to the program at the same time, this does not have to be the case and in fact some device operations are designed to present separate status interruptions. To handle this situation the program combines the status of multiple interruptions to determine the overall status of the channel and device. The `OC` does that by combining the current known status, in this case initially all zeros, with the status presented by the interruption. Chapter 16, pages 13-62 – 13-78, of the S/370 PoO details the content of the CSW and the various status conditions. The combined status is inspected for any error conditions, as follows.

```
CLI     STATUS+1,X'00'  Did the channel have, a problem?
BNE     CHNLERR        ..Yes, end with a channel error
```

The first check is for channel level errors. Many of the detected conditions are programming created error conditions. These are useful in debugging. Others relate to physical channel issues, none of which can happen with use of an emulator, such as Hercules. Nevertheless, because SATK strives to function in all mainframe contexts, the checks are still performed.

The next set of status conditions are reported as unit status. Some may be errors, some not depending on the device. For the console, all of these are considered as errors.

SATK User Guide

TM	STATUS,X'F3'	Did the unit encounter a problem?
BNZ	UNITERR	..No, end with a unit error

If none of these conditions happened, then all that is left is to determine if the I/O has completed. This requires both the Channel End and Unit End conditions to be set.

TM	STATUS,X'0C'	Did both channel and unit end?
BNO	DOWAIT	Wait again for both to be done

The test assumes that if either status (or both) have not been presented, they eventually will. The CPU is then caused to wait again by branching back to the DOWAIT label, and wait for an interruption. When both the Channel End and Unit End conditions have been signaled, the branch is ignored and the program falls through to the next instruction. The message has been successfully sent to the console.

	LPSW	DONE	Normal program termination
DONE	DWAITEND		Successful execution of the program

And that is it! The program is done. The PSW loaded has all PSW interrupt masks off and waits, a condition from which the CPU will not exit (except by a manually triggered restart, which of course was earlier trapped). The DWAITEND macro creates the appropriate architecture level PSW for the program to enter this termination state.

Step 1 – Assemble the Program

This program utilizes some macros supplied with the SATK in a macro library. Macro libraries are accessed via the environment variable MACLIB. This environment variable requires the SATK delivered macro library: `${SATK}/maclib`.

```
export MACLIB=${REPO}/maclib      # Access SATK supplied macros
${ASMA} -t s370 -d --stats -g ldip1/pgm3.txt -l asma-$sfx.txt pgm3.asm
```

The above statements are from the `asm` script in the `pgm3` directory. All that has really changed from the previous programs is the addition of the `export` statement that creates the `MACLIB` environment variable.

Step 2 – Create the IPL Medium

Again the `med` script is used to create the IPL medium for Program 3.

```
${IPLASMA} -v -f ld -m pgm3.3310 --records --asa=ASAREGN.bin \
ldip1/pgm3.txt >> iplasma-${sfx}.txt
```

The `iplasma.py` command line is very similar to the one provided for Program 2. However, a new argument is provided: `--asa=ASAREGN.bin`. This argument informs `iplasma.py` that a PSW region has been replaced by an assigned storage area region. The ASA region's file name, including its extension, is provided. This argument implies that the IPL PSW is at

SATK User Guide

the beginning of the region. The other arguments are the same (other than for the adjustment to the “pgm3” filenames).

The FBA sector allocation map has changed as a result of including the `--asa` argument.

```
FBA DASD Map:
  IPL0: 0-0
  VOLLBL: 1-1
  ASA: 2-2
  IPLPGM3.bin: 3-3
```

A new sector has been allocated for the ASA area. Why? Does it not contain the IPL PSW? Yes. But, the FBA sector zero must contain both IPL records 0 **and** 1. Record 1 contains the rest of the channel program that performs the IPL. So, the ASA content is moved to the second sector and the channel program is adjusted to read this sector. Where did the IPL PSW come from? It was extracted from the ASA binary image file, bytes 0-7, and placed at the start of IPL record 0 as required by the IPL process.

```
Memory Map:
  IPL0: 000000-000017
  ASA: 000000-0001FF
  CCW0: 000008-000017
  IPLPGM3.bin: 000300-00040F
  IPL1: 000628-000657
```

As can be seen from the memory map, the ASA region has been allocated to the first 512 bytes of memory, as is expected from the assembly. The areas in **bold** have been created by `iplasma.py` from its knowledge of the intent of the `ASALOAD` and `ASAIPL` macros. Bytes X'0'-X'17' constitute IPL Record 0 (sector 0 content in **bold**). Bytes X'18'-X'48' contain IPL Record 1 (sector 0 content in *italics*).

```
FBA sector 0
000000  00080000 00000300 02000610 40000200
000010  08000628 00000001 43000648 40000008
000020  42000300 40000200 43000650 40000008
000030  42000000 00000200 06000001 00000003
000040  06000001 00000002 00000000 00000000
...
0001F0  00000000 00000000 00000000 00000000
FBA sector 2
000000  00080000 00000300 00000000 00000000
000010  00000000 00000000 00000000 00000000
000020  00000000 00000000 00000000 00000000
000030  00000000 00000000 00000000 00000000
000040  00000000 00000000 00000000 00000000
000050  00000000 00000000 000A0000 00000018
000060  000A0000 00000020 000A0000 00000028
000070  000A0000 00000030 000A0000 00000038
...
0001F0  00000000 00000000 00000000 00000000
```

SATK User Guide

```
FBA sector 3
000000 05C0D207 0000C0B6 B722C086 4810C08E
000010 9D001000 4740C06E 4720C072 4710C076
000020 D2070048 C08A9C00 10004740 C06E4720
000030 C0724710 C076D207 0078C0C6 8200C0BE
000040 D2070078 C0CE4910 00BA4770 C07AD601
000050 C0900044 9500C091 4770C07E 91F3C090
000060 4770C082 910CC090 47E0C034 8200C0D6
000070 8200C0DE 8200C0E6 8200C0EE 8200C0F6
000080 8200C0FE 8200C106 80000000 00000398
000090 000F0000 00000000 090003A0 00000017
0000A0 C8859393 9640C281 998560D4 85A38193
0000B0 40E69699 93845A00 000A0000 00010008
0000C0 020A0000 00000000 00080000 00000340
0000D0 000A0000 00000038 000A0000 00000000
0000E0 000A0000 00030004 000A0000 00030008
0000F0 000A0000 0003000C 000A0000 00030010
000100 000A0000 00030014 000A0000 00030018
000110 00000000 00000000 00000000 00000000
...
0001F0 00000000 00000000 00000000 00000000
```

Sector 1, as before, is omitted. A volume identification record has not been supplied. Sector 2 has the full contents of the ASA.

FBA IPL Channel Program

The IPL **channel program** in Sector 0 does the following:

1. The **first 24 bytes** of Sector 0 is read into storage at absolute address 0 by the hardware. The IPL function does this by sending a Read IPL command, X'02', to the IPL device. An **implied CCW** can be thought of as residing at absolute address 0. This implied CCW is not physically placed at address 0, but the hardware functions as if it were. The contents of this implied CCW causes the channel hardware to function as if the CCW was:

```
02000000 60000018, or in assembler format,
CCW X'02',0,B'01100000',24.
```

This implied CCW, a Read IPL command, reads 24 bytes starting at absolute address 0, with the command chain and suppress any incorrect length indication flags set. Command chaining is required to cause the IPL function's channel commands to continue with the CCW at address X'8'. The suppress incorrect length indication is required because Sector 0 is actually 512 bytes in length, but the implied CCW is only going to read the first 24 of those 512 bytes. As a result, an incorrect length would be indicated, but is suppressed by the implied flag.

SATK User Guide

2. The channel program continues, and with it the IPL function, with the CCW at absolute address 8. This CCW re-reads **all 512 bytes** of Sector 0. The first sector is read into memory addresses X'610'-X'80F'. Now a complete copy of IPL Records 0 and 1 reside in memory. IPL Record 0, in regular text, is repeated at X'610'-X'627'. IPL Record 1 now resides at X'628'-X'658'.

```
000610 00080000 00000300 02000610 40000200
000620 08000628 00000001 43000648 40000008
000630 42000300 40000200 43000650 40000008
000640 42000000 00000200 06000001 00000003
000650 06000001 00000002 00000000 00000000
```

3. The last thing that IPL Record 0 does is pass control of the CCW chain to address X'628', this time ignoring IPL Record 0, and continuing with IPL Record 1 in the second complete Sector 0.
4. The IPL program is then read into memory by the CCW's at X'628'-X'637' (in *italics*). These two CCW's use the information at X'648'-X'64F' to identify the starting sector, 3, and the number of sectors, 1, that constitute the content of the program.
5. The last thing done in the IPL Record 1's continuation of the channel program is the reading of the ASA starting at absolute address 0. The underlined CCW's at X'638'-X'657' use the information at addresses X'650'-X'657' to read 1 sector starting with sector 2. Sector 2 is the ASA region. This wipes out the IPL PSW and the two CCW's of IPL Record 0 read into memory at the beginning of the IPL function.
6. At the conclusion of the IPL channel program CCW's, the IPL device address will be stored in the I/O Interrupt ID field at absolute X'BA'. A CSW will also be stored at X'40'. If an I/O error occurred, the IPL function ceases at this point and will not load the IPL PSW.
7. A successful IPL completes with the loading of the IPL PSW (now from the ASA region) and turning control over to the CPU.

Step 3 – Run the Bare-Metal Program

The `ipl` script will, as before, execute the program. This script has been adjusted for `pgm3` file names.

The Hercules configuration file contains these statements:

```
# Hercules sample configuration file for pgm3.asm
ARCHMODE S/370          # S/370 targeted by pgm3.asm
MAINSIZE 16K             # Sample starts at X'300'
NUMCPU 1
```

SATK User Guide

```
DIAG8CMD enable          # Allow use of DIAG X'008'

# Devices
000F 3215-C /            # Console required by mainframe systems
0110 3310 pgm3.3310      # FBA IPL disk
```

The only change has been to change the file name of the emulated FBA disk to Program 3.

The Hercules run control file, `pgm3.rc`, is as follows.

```
# This file is intended for use with Hercules Hyperion
# Comment either or both statements if you do not want to trace
t+
t+00F
# Perform the FBA based IPL
ipl 110
```

A single statement has been added since Program 2: `t+00F`. This statement causes tracing of the device, the console. In the Hercules log we can find this:

```
23:04:51 /Hello Bare-Metal World!
```

The `/` tells us that the content of the line came from the bare-metal program (and not Hercules) on the integrated console. There is our “Hello World” using mainframe channel I/O!