Hyperion Validation

Document Number jh16-0037-00

December 2, 2016 11:48 a.m.

John P. Hartmann

First Edition (November 2016)

Contents

Preface

This document describes the tools and environment to run test cases to validate Hyperion on a UNIX system, such as Linux, FreeBSD, or Mac OS/X.

The Windows environment is similar, except that a test is executed by `runtest.cmd` rather than `runtest`.

Terminology

A few terms in this document have specific meanings and mixing them up will confuse you.

Command script. A list of UNIX or Windows commands that are processed by the operating system's command processor, such as `sh`, `bash`, or similar; and Windows `command.com`.

runtest. This is surely overloaded:

- The `runtest` command is a shell script that is processed by a UNIX shell, such as `sh` or `bash`.
- The `runtest` script command; it starts Hyperion executing instructions in a test case.
- A *CMS Pipelines* filter, *runtest*, that runs a test case on z/CMS.

Script command. A command that is interpreted by the Hyperion script console command. Script commands cannot be issued from the console panel.

Hyperion enhancements in support of testing

The `-t` flag on the Hyperion command runs the script command specified by the `-r` flag as a coroutine: Hyperion and the script take turns at running.

Control transfers to Hyperion instruction simulation by the `runtest` script command; control reverts to the script when all CPUs are in the stopped state, but the state of the test program remains and it can be resumed by another `runtest` script command.

Prerequisite software

You need a REXX interpreter to run the post processor. A system/360 assembler is a more than nice to have, but strictly not a requirement.

REXX interpreter

A REXX interpreter must be installed in your system, but it need not be configured in Hyperion.

On Windows (and even AIX) you may be fortunate to have IBM's `ObjectREXX` installed even if it was withdrawn in 2004; several bugs that it took me 10 years to get fixed have regressed in `ooRexx`.

If you are not that fortunate, two popular interpreters are available for your choice, according to taste. You will likely find precompiled packages for your distribution.

Open Object Rexx

Fondly known as `ooRexx`.

`http://www.oorexx.org/`
`https://sourceforge.net/projects/oorexx/`

It is an open source derivative of a subset of the IBM program product Object REXX (the parts that IBM owns).

Notes:

1. Do not use with pipes or sockets; you get random premature end-of-file.

   This was a design error in Rick McGuire's original OS/2 implementation that is due to an assumption that `read()` of a socket works the same as for a file (or ignorance of sockets, perhaps).

   I eventually got it fixed in Object REXX; and to my delight, the fix was carried forward into `ooRexx` only to be regressed by an ignoramus who decided to "improve" on the library.

Me? I wrote my own REXX classic when `ooRexx` became untenable for me.

Regina Rexx

`http://regina-rexx.sourceforge.net`. This implementation predates `ooRexx`; it was the implementation of choice before that.

Assembler

While not strictly required, having an assembler as opposed to hand assembly is so much more productive that realistically, it is a requirement.

Several assemblers are available and run on various platforms. My recommendation is that you use ASMA, which is part of Harold Grovesten's SATK.

Except for ASMA, you will need a utility to convert TEXT to a core image file. A sample is present in `tests/mkcore.rexx`.

Notes:

1. Unless you have a HLASM compatible assembler on the workstation or you do require features the ASMA does not implement, ASMA should be your assembler of choice, particularly because it is free and runs just about everywhere.
2. Keep in mind that others might not be able to enhance or fix your test case if it requires a proprietary assembler.

ASMA from SATK (stand alone toolkit).

ASMA is the simplest one to use and integrate, though it has a number of limitations when compared to HLASM.

As ASMA is written in Python, it is readily available wherever Python is available, which certainly includes all platforms on which Hyperion can be installed.

Refer to Appendix A, “Using ASMA to assemble testing programs” on page 22 for installation and use information.

Notes:

1. SATK requires Python 3.3, which is normally preinstalled in Linux distributions.

z390 portable assembler

This is a conglomerate that includes a HLASM compatible assembler.

Allegedly, it does not support privileged instructions, so in this context it is a non-starter.

`http://www.z390.org/`

CMS or MVS assemblers

This, of course assumes that you have access to such a system.

If all fails, VM/370 release 6 under Hyperion might be usable with the macro library to define z operations:

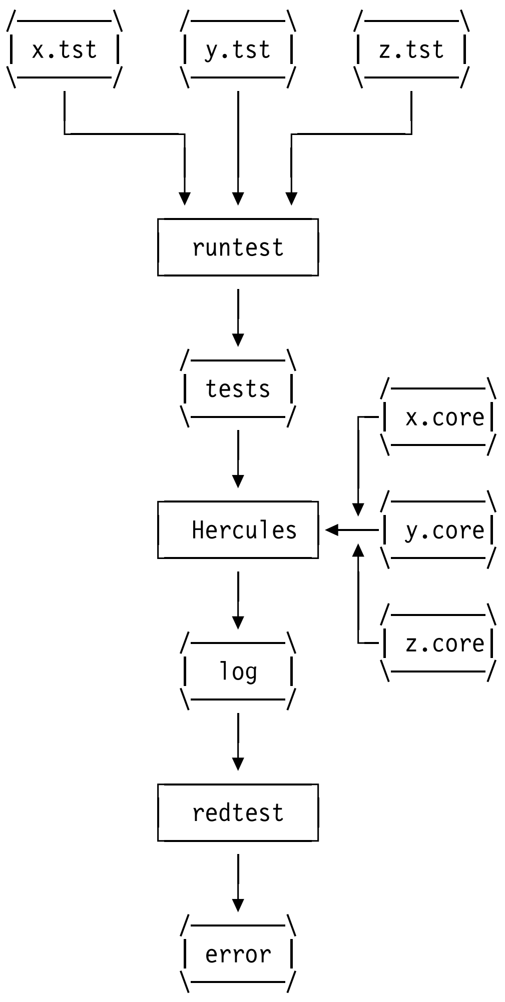
`http://vm.marist.edu/˜pipeline/zops.copy`

Having created a TEXT deck and a listing, you download to your workstation and use `mkcore.rexx` to create the core image.

Introduction

One runs test cases to verify correct operation of Hyperion. Such tests are also called a validation suite or use cases.

Work flow



The mechanics of running a test case are simple; they are embodied in the `runtest` command:

1. Generate a composite input file by file name expansion of the input test script names.
2. Start Hyperion specifying the test case script with the `-r` and `-t` command flags and redirect standard output to a file.
3. The command script would typically load an executable core image and start it running using the `runtest` script command (that is, not a recursion of the `runtest` shell script).

   The testing script regains control when all processors are in the stopped state.
4. Hyperion runs the test script to completion.

5. Pass the output file through `redtest.rexx` to process testing directives in the file. Testing directives are loud comments as seen by Hyperion; they are described in "Using `redtest.rexx` directives" on page 9.

The net result is pass or fail.

The one thing to remember here is that testing directives are comments as far as Hyperion is concerned; there is no way testing directives can modify Hyperion behaviour.

make check

Having built, but not installed Hyperion, you can issue `make check` to run all test cases in the `tests` subdirectory.

You should receive a message that *n* test cases have been done and all pass.

Preparing a test case

Whatever you wish to test you will need a script of Hyperion console commands. Make its file name extension `.tst`. If there is any chance it gets near z/CMS, the file name must be unique within the first eight characters.

If you are testing the Hyperion command processor, this is all you need, but normally you will want some code for the test to run.

Test script

The test script contains:

1. Comments, quiet or loud.
2. Console commands to set up the test case and to display the results.
3. Test file directives, which are loud comments as far as Hyperion is concerned, but control the `redtest.rexx`. Directives are a word prefixed an asterisk, such as `*Testcase`.

Executable

You can create the program to run, if any, by storing into real storage using `r` console commands, but that is an error prone and labour intensive process; most testers use some kind of assembler.

ASMA is the assembler for you unless you have access to MVS or z/CMS; and even so, ASMA might be more expedient than downloading from the host. Care and feed of ASMA is described in Appendix A, “Using ASMA to assemble testing programs” on page 22.

The output deck from the assembler must be converted to a core image file unless you use ASMA. `test/mkcore.rexx` is a utility for such conversion.

Make file rules

These rules might help compiling your test cases, though any HLASM activity is likely to happen on z/VM:

```
.SUFFIXES: .text .tst .assemble .rexx .asm .core .out .comp

%.core: %.asm
   ASMPATH=. ${APATH}/asma.py -l $*.list -i $@ $<

%.core: %.assemble
   hlasm -option term -o . -targetascii $<
   rexx mkcore.rexx <$*.text >$@
   rm $*.text

%.tst: %.assemble
   hlasm -option term -o . -targetascii $<
   rexx text2tst.rexx <$*.text >$@
   rm $*.text
```

Mechanics

In its simplest form the test case:

1. Sets up architecture mode and the contents of storage, including the restart PSW.
2. Fires up the test program by the `runtest` script command.
3. Displays execution results by console commands when it regains control; and verifies the results by suitable `redtest.rexx` directives.

However, a test case can invoke the program under test several times; the cipher test case is a sterling example of this.

Example test case

A simple test that does not run on z/CMS.

The program:

```
* Multiply halfword immediate test

* This file was put into the public domain 2015-10-22
* by John P. Hartmann.  You can use it for anything you like,
* as long as this notice remains.

mhi start 0
 using mhi,0
 org mhi+x'60' Unused bcmode stuff as scratch
stop dc x'00020000',f'0',ad(0)
 org mhi+x'1a0' Restart
 dc x'0000000180000000',ad(go)
 org mhi+x'1d0' Program
 dc x'0002000180000000',ad(x'deaddead')
 org mhi+x'200'
go equ *
 la 2,x'21'
 lr 5,2
 mhi 2,2
 lr 6,2
 ipm 4
 la 3,1(2,2)
 lpswe stop
 ltorg
 end
```

The test script:

```
* This file was put into the public domain 2015-10-22
* by John P. Hartmann.  You can use it for anything you like,
* as long as this notice remains.

*Testcase mhi
sysclear
archmode z
loadcore "$(testpath)/mhi.core"
runtest .1
*Compare
gpr
*Gpr 2 0042
*Gpr 3 0085
*Gpr 4 0000000000000000
*Done
```

The variable `testpath` that is referenced by the `loadcore` console command is set by `runtest` command; it is stored in the Hyperion variable pool.

Running the test case from a source directory:

```
[/home/john/src/hyperion] ./runtest -f mhi -d . -h /usr/data/src/hercules/dbg/
Files: ./mhi.tst
Variable ptrsize is set to "8".
Variable platform is set to "Linux".
Test mhi.  4 OK compares.  All pass.
Done 1 tests.  All OK.
```

`runtest`—Shell script to run test cases

The `runtest` command or `runtest.cmd` batch file on Windows runs `.tst` scripts to verify proper Hyperion functioning.

Issue `make check` in the directory where you built Hercules to run the regression test. `runtest` command first creates a composite set of test cases (`allTests.testin` in the current working directory) and then invokes Hyperion to run them using the configuration file `tests.conf` and the file just created. When it completes, the console output (`allTests.out`) is then inspected by `redtest.rexx`. and the result is reported to the user.

The `runtest` commandt return code is the number of failed test cases.

`runtest` command

```
               ┌───────────────┐
►►──runtest──▼─┴───────────────┴──────────────────────────────────►◄
               ├─-d──word──────┤
               ├─-e──word──────┤
               ├─-f──word──────┤
               ├─-h──word──────┤
               ├─-l──word──────┤
               ├─-p──word──────┤
               ├─-q────────────┤
               ├─-r──number────┤
               ├─-t──number────┤
               ├─-v──word──────┤
               ├─-w──word──────┤
               └─-x────────────┘
```

`-d`	Specify the path to the test directory, that is, the directory that contains the `.tst` files. The default is `${dirname $0}`, perhaps `../hyperion/tests`.
`-e`	Specify the default extension for the input test files; the default is `.tst`. The `-e` flag is active for all `-f` flags until the next `-e` flag, if any.
`-f`	Specify the input file name perhaps including an extension. The default is `-f *.tst` to run all test cases in the test directory. Multiple `-f` flags are supported. The default extension is appended if the word does not contain a period.
`-h`	Specify the object directory that contains the Hyperion executable. The default is parent of `-d` if there is no Hyperion executable in the current directory (that is, in-source build).
`-l`	Specify the name of a module to load dynamically at start. This is passed to Hyperion.
`-p`	Specify the library path for loadable modules, *i.e.*, device managers. This is passed to Hercules.
`-q`	Pass `quiet` to `redtest` to suppress details about test cases.

`-r`	Repeat the composite test script *n* times. There is no need to specify `-x` as Hyperion terminates at end of file on the input file without an explicit quit when in daemon mode.
`-t`	Passed to Hyperion (timeout factor).
`-v`	Set variable for `redtest.rexx`. Use `-v ptrsize=4` when running a 32 bit Hyperion executable if no valid `config.h` can be found. You can also set private variables this way (but such variables will be unset when running `make check`). It is unspecified whether case is respected in variable names. **Note:** This variable pool is separate from the variable pool that you can reach with the `defsym` console command, which refers to the Hyperion variable pool.
`-w`	Specify the work file name. The default is `allTests`.
`-x`	Do not append the quit command to the composite test script.

runtest—Script command to switch to executable code

The test a `.tst` script defines is normally begun via the `runtest` script command which supports optional arguments indicating how the test should be started as well as the maximum amount of time it should run.

`runtest` script command waits only until all CPUs are in the stopped state.

```
          ┌─restart──────┐   ┌─30──────┐
►►──runtest──┼──────────────┼───┼─────────┼──┬────────────────┬────────────►◄
          ├─start────────┤   └─timeout─┘  └─# comment...─┘
          └─┤ oldpsw ├───┘
```

oldpsw:

```
├──┬─external─┬──────────────────────────────────────────────────────────┤
   ├─svc──────┤
   ├─program──┤
   ├─machine──┤
   └─io───────┘
```

The first argument is optional. If specified, it must be either `restart`, `start`, or an interrupt type. It specifies how the test should be started: via the Hercules `restart` console command or the `start` console command. The default is to start the test via the `restart` command.

When an interrupt type is specified, the old PSW for that type is copied to the restart new PSW before the restart command is issued to fire up the test case where it left off taking an interrupt for which the new PSW was disabled.

The second argument, which is also optional, defines the maximum amount of time (specified in a whole or fractional number of seconds) that the test is allowed to run. This prevents runaway due to a bug in the test or within Hercules itself. Under normal circumstances all tests should end as soon as all CPUs are in the stopped state. The default timeout is 30 seconds.

Notes:

1. The first execution after a core file has been loaded is normally kicked off by `restart`; using the `psw` console command to set up the current PSW is rather fiddly, but if you do, `start` is appropriate.
2. A test case can stop temporarily for the test script to check results along the way. Execution should be resumed by `start` when execution was suspended by signal processor stop; it should be resumed from the old PSW when a disabled new PSW was loaded, perhaps for an SVC interrupt.
3. Timeout values are multiplied by the test timeout factor value which is specified by the `-t` switch on the Hyperion command. The test timeout factor value allows for running tests on systems that may be slower than the system they were originally designed for.
4. The `runtest` script command is usually followed by Hyperion console commands to display storage or registers, *etc*, and test directives described in “Using `redtest.rexx` directives” on page 9 to verify the test results.

Using `redtest.rexx` directives

Test script (`.tst`) files are executed as Hyperion `.rc files`, so they contain Hyperion console commands. A number of loud comments (beginning with an asterisk) are interpreted by `redtest.rexx` as directives. All directives are case sensitive.

For example, `*testcase` accomplishes nothing (other than get you an error).

Predefined variables

Predefined variables are read only. To reference them, specify the variable name prefixed a dollar sign.

The values for predefined variables are gathered from several sources:

- The `-v` command flag on `runtest` command which is passed as arguments to `redtest.rexx`.
- They are extracted from the list of features in the features message, `HHC01417I`. you can test, for example:

  `*If $cmpxchg1 = 1`

- Logic internal to `runtest` command.
  - The output of `uname -s` is assigned to `platform`. On windows, its value is hardwired `Windows`.
  - When the `runtest` command finds `config.h`, it extracts the size of a pointer and sets the variable `ptrsize` to 4 or 8, as appropriate.

    If you receive a message to the effect that `config.h` is not found, you should set the variable explicitly using the `-v` flag.

Conditional directives

You can issue directives conditionally by including them in an if/then/else block.

```
mainsize 2147483648b
*If $ptrsize = 4
*Error 1 HHC01430S Error in function configure_storage(2G): Cannot allocate memory
*Error   HHC02388E Configure storage error -1
*Else # 64 bit
*Info HHC17003I MAIN     storage is 2G (mainsize); storage is not locked
*Fi
```

You can even nest such blocks, for example to cater for differences between UNIX and Windows:

```
mainsize 2147483647B
*If $ptrsize = 4
*If $platform = "Windows"
*Error 1 HHC01430S Error in function configure_storage(2G): Not enough space
*Else # e.g. Linux
*Error 1 HHC01430S Error in function configure_storage(2G): Cannot allocate memory
*Fi
*Error   HHC02388E Configure storage error -1
*Else # 64 bit
*Info    HHC17003I MAIN     storage is 2G (mainsize); storage is not locked
*Fi
```

Do not include console commands inside such a block as Hyperion does not understand of the conditional nature of the directives, which are executed after Hyperion has finished. Put another way, if you include console commands in both branches of an if/then/else, all commands are processed by Hyperion; not likely to be what you desire; even if you do it right, it is likely to confuse the unsuspecting tester.

Alphabetical list of `redtest.rexx` directives

`*`—Comment

The comment is ignored. There must be a blank after the asterisk to avoid the comment being interpreted as a directive.

`*Compare`—Start verification section

The output from a subsequent `r` console command is stored for comparison by a `*Want` directive. Only the last line of a multiline display is stored. You would normally issue `r` and `*Want` in pairs. If you issue an unpaired `r` console command, the output is stored for documentation in the output file where you can inspect it, but `redtest.rexx` ignores it.

`*Done`—End a test case

```
►►──*Done──┬────────┬──────────────────────────────────────────────────►◄
           └─nowait─┘
```

End of a test case. Specify `nowait` if the test case does not end by loading a disabled PSW, for example, when it tests just console commands without running any code.

`*Else`—Specify conditonal alternatives

Flip the setting of the matching `*If` directive condition, unless nested in a suppressed `*If` directive.

`*Error`—Verify a Hyperion panel message

Synonym for `*Hmsg` directive. Use if an error message is expected.

`*Explain`—Explanatory text

Specify explanatory text to write when the next comparison fails.

You can specify as many `*Explain` directives as you like for multiline messages. The explain array is cleared once a test has been done, whether OK or not.

`*Fi`—Close block of conditional directives

End an `*If` directive. The process/ignore setting for directives is unstacked.

`*Gpr`—Verify general register contents

```
►►──*Gpr──number──hex──┬──────────┬──────────────────────────────►◄
                       └─#address─┘
```

A previous `gpr` console command must have been issued to display the general registers.

The specified register is compared with the hexadecimal data specified after the register number. Specify a decimal register number with no leading zero.

The comment "#address" may be specified to notify the CMS test driver that the register contains a storage address rather than binary data. When specified, the base address of the simulated absolute storage is factored out of the comparison.

`*Hmsg`—Verify a Hyperion panel message

```
►►──┬─*Hmsg──┬──┬────────┬──messageId──string─────────────────►◄
    ├─*Imsg──┤  └─number─┘
    └─*Error─┘
```

Specify the complete informational message string to compare against the last issued message, except for messages that are processed by `redtest.rexx`, and messages 7 and 1603.

A number is optional before the message identifier to specify the last but one, last but two, etc, see example below.

Note: The difference between the three directives is only in the message issued when a message does not match; it can be considered cosmetic.

`*If`—Open block of conditional directives

```
►►──*If──booleanExpression───────────────────────────────────►◄
```

Specify a condition to test. The expression is evaluated by the REXX interpreter after substituting any variables, which are specified by a dollar prefix to and identifier (these are not REXX variables).

If the condition holds, the following directives are processed up to the matching `*Else` directive or `*Fi` directive. If the condition does not hold, nested directives are ignored except that nested `*If` directives are parsed to ensure that nesting is processed correctly.

`*Info`—Verify a Hyperion panel message

Synonym for `*Hmsg` directive. Use if an informational message is expected.

`*Key`—Verify storage key

```
►►──*Key──hex────────────────────────────────────────────────►◄
```

Compare the specified key against the key displayed in the previous `r` console command.

Specify two hexadecimal digits. Data that are compared include fetch protect, reference, and change bits.

`*Message`—Print message

Print a message on the output from `redtest.rexx`; not to the input file, which is the output from Hyperion.

`*Prefix`—Verify contents of prefix register

```
►►──*Prefix──hex──────────────────────────────────────────────►◄
```

Compare the specified value to the contents of the prefix register displayed by a previous `pr` console command.

`*Program`—Verify program interrupt code

```
►►──*Program──hex─────────────────────────────────────────────►◄
```

A program check is expected, for example to verify that a privileged operation causes an 0c2, which is specified as `*Program 0002`.

Notes:

1. `redtest.rexx` obtains the program check interrupt code from message `HHC00801I`, which is not issued when the `ostailor quiet` console command is active.
2. This directive must be specified before the `runtest` script command that starts the test case.
3. When your program contains its own program check handler you have two choices assuming that you are testing whether program checks are recognised as required. You can use the `ostailor quiet` console command to suppress the console message that triggers `redtest.rexx` into complaining; or you can specify `*Program` directive to fake it to `redtest.rexx` that the program check message is what you expect.

   Truly “two choices” in American idiom.

`*Testcase`—Begin a test case

```
►►──*Testcase──string─────────────────────────────────────────►◄
```

Counters are reset appropriately.

`*Timeout`—Boiler-room test

A timeout is expected. Do not use.

`*Want`—Verify storage contents

```
►►──*Want──┬────────────┬──hexdata───────────────────────────►◄
           └─"string"───┘
```

Specify the expected `r` console command output in hexadecimal. A string identifying the comparison may be enclosed in double quotes before the compare data.

Idioms

You should test a general instructions in problem state, but it could be argued that it also needs verification in supervisor state. And then there are the three architecture modes (assuming the instruction can trace its pedigree back to System/360) and the three addressing modes.

A thorough tester is truly an ugly customer.

Testing in problem state

Running in problem state is accomplished by turning on bit 15 in the restart PSW.

The immediate challenge is how to stop the program when it is done, but does not fail.

Recall that the criterion for ending a `runtest` script command is that all CPUs are in the stopped state.

You can put a Hyperion CPU into the stopped state by:

- Executing the `SIGP` stop instruction. This is the only way on real iron.
- Loading a disabled wait PSW. `LPSW` and `LPSWE` clearly are capable of this.
- Entering a program interrupt loop. Not particularly attractive.
- And the only thing you can do in problem state: recognising an SVC interrupt for which the new PSW is disabled.

So the solution is to terminate a problem state program by an SVC instruction with a suitable new PSW.

Let us make the convention that SVC 0 indicates that the program is terminating normally (not an EXCP). The other 255 SVC codes can be put to action as indications of errors.

Verification ensures that location X'88.4' contains X'00020000'. Note that this verifies that an SVC was issued by testing the instruction length code as well as the SVC code.

The source code:

```
prob     TITLE 'Null problem state test case.'
*
* This file was put into the public domain 2016-11-29
* by John P. Hartmann.  You can use it for anything you like,
* as long as this notice remains.
*

prob start 0
 using prob,15
 org prob+x'1a0' Restart
*        P
 dc x'0001000180000000',ad(go)
 org prob+x'1c0' SVC new
 dc x'0002000180000000',ad(0)
 org prob+x'1d0' Program
 dc x'0002000180000000',ad(x'deaddead')
 org prob+x'200'
go equ *
 svc 0 OK
 end
```

The test file:

```
*
* This file was put into the public domain 2016-11-29
* by John P. Hartmann.  You can use it for anything you like,
* as long as this notice remains.
*

* Null problem state program for education

*Testcase problem
sysclear
archmode z
loadcore "$(testpath)/problem.core"
runtest .1
*Compare
r 88.4
*Want 00020000
*Done
```

Running it:

```
[/home/john/src/hyperion] ./onetest problem
Files: ./problem.tst
Variable ptrsize is set to "8".
Variable platform is set to "Linux".
Test problem.  2 OK compares.  All pass.
Done 1 tests.  All OK.
```

Testing privileged instructions

Testing a privileged instruction should first of all verify that it does cause an 0c2 in problem state.

Having verified that, testing of the instruction execution should proceed.

This means that we shall start the test in problem state and then resume it in supervisor state. We nuke the problem state bit in the problem old PSW to get out of the problem state jail.

To be even more illustrative, the test case first verifies an 0c1 for `ISK`, then 0c2 for `ISKE`, and then tests `ISKE` for real.

The source code:

```
PRIVOP   TITLE 'Null (well almost) test case for a privop'
*
* This file was put into the public domain 2016-11-29
* by John P. Hartmann.  You can use it for anything you like,
* as long as this notice remains.
*
PRIVOP start 0
 using PRIVOP,15
 org PRIVOP+x'1a0' Restart
 dc x'0001000180000000',ad(go)
 org PRIVOP+x'1c0' SVC new
 dc x'0002000180000000',ad(0)
 org PRIVOP+x'1d0' Program
 dc x'0002000180000000',ad(x'deaddead')
 org PRIVOP+x'200'
go equ *
 isk 0,0
* Stop by disabled program check
 iske 0,0
* Stop by disabled program check
 la 0,x'fff'
 iske 0,0
 svc 0
 end
```

The test file:

```
*
* This file was put into the public domain 2016-11-29
* by John P. Hartmann.  You can use it for anything you like,
* as long as this notice remains.
*
*Testcase privopisk
sysclear
archmode z
loadcore "$(testpath)/privop.core"
*Program 1
runtest .1
*Done

*Testcase privopiske
*Program 2
runtest program .1
*Done

*Testcase privopgo
r 151=00
runtest program .1
*Compare
gpr
*Gpr 0 0000000000000f06
r 88.4
*Want 00020000
*Done
```

Running it:

```
Files: ./privop.tst
Variable ptrsize is set to "8".
Variable platform is set to "Linux".
Test privopisk.  2 OK compares.  All pass.
Test privopiske.  2 OK compares.  All pass.
Test privopgo.  3 OK compares.  All pass.
Done 3 tests.  All OK.
```

Notes:

1. The domain of the `*Program` directive is to the next `*Done` directive. That means that we must split this test case into three `*Testcase` directive sections.
2. The use of `runtest program` to continue after the disabled waits.

Testing a semi-privileged instruction

As an example, consider SPKA. In problem state, the key is set only when the corresponding bit in control register 3 is set.

The program starts by loading control register 3; we could also have achieved this by the `cr` console command. It then sets key 8, which is forbidden in problem state, but as we are in supervisor state, nothing untoward happens; it then saves the key in register 4 for verification. To continue, it switches to 24-bit in problem state key 15, sets key 4 and attempts to set the key to the forbidden 8, which should cause a privileged operation exception. In case it does not, an SVC 1 acts as a back stop.

Were an IPK attempted in problem state, it would fail here because extract authority is not enabled.

Verification includes checks for no SVCs interrupts fielded; correct program old PSW and the fact that the program ended by a privileged operation interrupt.

The source code:

```
SEMIPRIV TITLE 'Test a semiprivileged instruction'
*
* This file was put into the public domain 2016-11-30
* by John P. Hartmann.  You can use it for anything you like,
* as long as this notice remains.
*
SEMIPRIV start 0
 using SEMIPRIV,15
 org SEMIPRIV+x'1a0' Restart
 dc x'0000000180000000',ad(go)
 org SEMIPRIV+x'1c0' SVC new
 dc x'0002000180000000',ad(0)
 org SEMIPRIV+x'1d0' Program
 dc x'0002000180000000',ad(x'deaddead')
 org SEMIPRIV+x'200'
go equ *
* Supervisor state part
 lctl 3,3,cr3
 spka x'80' Forbidden in problem state, but not for me
 ipk , Current key -> r2
 lgr 4,2 Save proof
 sgr 0,0
 lpsw prob
goprob ds 0h
 spka x'40' Should be OK
* ipk here would privop since CR0.36 is 0.  Test separately.
 spka x'80'
 svc 1 If we survive

prob dc 0d'0',x'00f9',h'0',a(goprob) Problem state and 24-bit
cr3 dc a(x'7f7f0000') disallow key 0 and 8
 end
```

The test file:

```
*
* This file was put into the public domain 2016-11-30
* by John P. Hartmann.  You can use it for anything you like,
* as long as this notice remains.
*
*Testcase semipriv
sysclear
archmode z
loadcore "$(testpath)/semipriv.core"
*Program 2
runtest .1
*Compare
gpr
*Gpr 4 0000000000000080
# No SVCs
r 88.4
*Want 00000000
# key 4 and problem state in prog old PSW
r 150.2
*Want 0041
*Done
```

Running it:

```
Files: ./semipriv.tst
Variable ptrsize is set to "8".
Variable platform is set to "Linux".
Test semipriv.  5 OK compares.  All pass.
Done 1 tests.  All OK.
```

Who said that System/370 was simple?

Going wild

So far, we have stuck to the model of a `.tst` file that contains console commands and a `.asm` file that contains a program that is converted to a core image by ASMA and loaded into storage by the `loadcore` console command. This paradigm will serve you well until things get real hairy.

Program fragment for dissection

For the following discussion, consider this partial program, the beginning of which you have seen as boiler plate in previous chapters, but now enhanced to support multiple invocations:

```
 org x'1a0' Restart new
rstnew dc x'0000000180000000',ad(go)
 org x'200'
go ds 0h
 <do something>
 svc 0 OK so far
resume org rstnew
 dc x'0071000000000000',ad(next) Key 7 24 bit
 org resume
next ds 0d
 <do something more>
 svc 1 Not an error, but to distinguish the two
```

This program fragment could be a paradigm of entering a test under varying conditions where each run is kicked off by its own `runtest` script command.

ASMA reaction

This would fail abysmally because the core image would contain the final contents of the `rstnew` PSW rather than two.

Back to the drawing board. (What is this man doing? Trying to reverse entropy?)

HLASM reaction

On the face of it, this is not useful since we obtain an object module rather than a core image.

But that is easily corrected by a simple REXX program that obtains the program text from what is known as `TXT` records and converts this text to suitable `r` console commands (and throws away lots of other information in the process).

The rub here is that HLASM produces a number of `TXT` records for the sample program:

1. The initial contents of `rstnew`.
2. The first program at label `go`.
3. The second contents of `rstnew`.
4. The second program at label `nest`.

Actually, HLASM flushes its text buffer when it assembles one of these assembler instructions: `CSECT LOCTR ORG PUNCH REPRO`.

Effectively, we have produced a core image when running the `r` console commands produced by `text2core.rexx`.

Were we to stop here, all we would have achieved is an obfuscation of the way ASMA creates a core file. We also have not produced any commands to run the program; were we to do that after the sequence of `r` console commands, we would be no better off than with ASMA.

This is how we can do better: HLASM supports two assembler instructions `REPRO` and `PUNCH` that emit your text into the object deck, *at the current position*; and that is the rub.

That is, when processing one of those two instructions, HLASM first flushes its output text buffer into a `TXT` record. It then punches whatever we told it to into another 80-byte record, padding with blanks as required, and finally resumes collecting object code in a third `TXT` record.

So let us resume the sample program above after the the first SVC:

```
 svc 0
 punch 'runtest .1'
 punch '*Compare'
 punch 'r 88.4'
 punch '*Want 00020000'
 punch '*Done'
resume org rstnew
```

The utility to generate `r` console commands will need change to pass anything that does not begin X'02' through to the output unmodified. You would also punch the initial boiler plate at the very beginning of the program, but we did not show that to keep you in suspense.

Make note that we start instruction processing *before* the remainder of the program is even loaded into core. Clearly, you will need to ensure that all you need has been assembled at the point you start injecting commands into the object stream.

So the start of the program could be (the real code):

```
* These are punched before the ESD record(s)
 punch '*Testcase multix'
 punch 'sysclear'
 punch 'archmode z'
mlt start 0
 org mlt+x'1a0' Restart new
rstnew dc x'0000000180000000',ad(go)
 org mlt+x'1c0' SVC new
 dc x'0002000180000000',ad(0)
 org mlt+x'200'
go ds 0h
 l 0,0 <do something>
 svc 0 OK so far
 punch 'runtest .1'
 punch '*Compare'
 punch 'r 88.4'
 punch '*Want 00020000'
 punch '*Done'
resume org rstnew
 dc x'0071000000000000',ad(next) Key 7 24 bit
 org resume  Back; org (no operands) likely works too
next svc 1
 punch '*Testcase multix1'
 punch 'runtest .1'
 punch '*Compare'
 punch 'r 88.4'
 punch '*Want 00020001'
 punch '*Done'
 end
```

And after the utility you have:

```
# This test file was generated from offline assembler source
# by txt2tst.rexx  1 Dec 2016 11:41:00
# Treat as object code.  That is, modifications will be lost.
# assemble and listing files are provided for information only.
*Testcase multix
sysclear
archmode z
r    1A0=00000001800000000000000000000200
r    1C0=00020001800000000000000000000000
r    200=580000000A00
runtest .1
*Compare
r 88.4
*Want 00020000
*Done
r    1A0=00710000000000000000000000000206
r    206=0A01
*Testcase multix1
runtest .1
*Compare
r 88.4
*Want 00020001
*Done
```

And it does run:

```
Files: ./wild.tst
Variable ptrsize is set to "8".
Variable platform is set to "Linux".
Test multix.  2 OK compares.  All pass.
Test multix1.  2 OK compares.  All pass.
Done 2 tests.  All OK.
```

Discussion—the fine print

- This is one-stop shopping; the only input is the `.assemble` file; no separate `.tst` needed, though you are free to embellish the output as you wish.
- You will find the utility to process a TEXT deck in `tests/text2tst.rexx`.

- We may to some extent have reversed entropy, but we did not engage in time travel: The entire command script is still run through Hyperion before any of the testing directives are processed, even though we have achieved the desirable goal of interspersing what produces test output with what checks it.
- Once ASMA can produce object decks, we we shall have no need to resort to another assembler.
- If you use an assembler on z/CMS or similar, the conversion by `text2tst.rexx` must happen in the EBCDIC domain and the output file converted to ASCII, perhaps during download. If assembled on the workstation, the assembler must produce ASCII output for `PUNCH` and `REPRO`.

And finally: This is where your journey will end. Interestingly, it was where mine began.

Appendix A. Using ASMA to assemble testing programs

ASMA is the simplest one of the assemblers to use and integrate, but it has some limitations as detailed below.

Installation

The code lives at:

```
https://github.com/s390guy/SATK
```

Clone from:

```
git@github.com:s390guy/SATK.git
```

No further action is required to install SATK; all you need is the path to the start-up Python script.

Using ASMA

If SATK is cloned into `/usr/data/src`, a make file could contain these actions to assemble a program:

```
.SUFFIXES: .asm .core
USD:=/usr/data/src
APATH:=${USD}/SATK/tools

%.core: %.asm
   ASMPATH=. ${APATH}/asma.py -l $*.list -i $@ $<
```

The recipe above sets the environment variable `ASMPATH` to the current directory and then invokes ASMA with the `-l` flag to generate a listing and the `-i` flag to generate a core image file.

Caveats

While ASMA so obviously is superior to hand assembly, it does have a few limitations relative to HLASM that you should be aware of.

- Always check the documentation of the assembler instructions you intend to use to be sure they are indeed implemented. Notable omissions are `AMODE` and `RMODE`, but these instructions are ignored, so you do not need to worry.
- Floating point constants are somewhat sketchy.
  - D'...' type floating point assembles as if FD'...' were specified, which is correct only for D'0'.
  - `DD`, `ED`, and `LD` work for decimal floating point constants.
  - All other floating point types are not supported.

  - `tests/dc-float.asm` contains, in no particular order, a number of binary floating point constants suitable for cut-and-paste into X type constants. This file was generated from the HLASM listing file from an assembly of float constants.

- Input is variable length ASCII. Continuation is specified by a backslash at the end of a line.

  Note: If creating a test case to run on both Hyperion and z/CMS, you must pay attention to any comment text sliding into column 72; ASMA will not warn you as it supports an infinite line length.

- You cannot generate a standard OS/360 object module; in fact, you cannot generate any form of relocatable code. However, that is not a limitation in this context.

Then there is a number of incompatibilities that I shall classify as nits:

- All DC operands must have nominal values. HLASM requires a nominal value only on constant operands that have a nonzero duplication factor.

I have noted a number of assembler instructions that, while sharing the mnemonic operation code with HLASM, require different operand formats:

COPY ASMA requires the complete file name, including extension, enclosed in quotes. As the OS/360 assemblers use `PDS` (Partitioned data set) for storage of macros and copy text, the name space is a single file name (as ASMA does for macros).

As ASMA is under active development, always consult the manual for the word from the horse's mouth. Do remember to `git pull` first.

Appendix B. Running test cases on z/CMS

You can verify a test case against real iron by running it on z/CMS; or more likely, by having a friend who will run the test case for you. Clearly, the z/CMS system must run under z/VM on a real z/Architecture machine.

This is accomplished by passing the `.tst` file through the *runtest* filter of the `CMSDRIVE` filter package.

As an absolute core file is of no use on z/CMS, the original test program must be assembled with something (HLASM or z390) that can generate OS/360 object code.

Supported console commands

The following console commands are supported; other attract a diagnostic.

```
*          #
ARCHLVL    ARCHMODE    GPR        LOADCORE
OSTAILOR   R           RUNTEST    SYSCLEAR
T+
```

`OSTAILOR` and `T+` are ignored. `ARCHLVL` verifies that z/CMS is running in the required mode; it cannot switch modes in flight.

`LOADCORE` loads the program using the z/CMS `LOAD` command.

Supported directives

The following directives are supported. In particular, variables and conditionals are unsupported.

```
*Compare   *Done       *Gpr       *Message
*Program   *Testcase   *Want
```

For the `*Gpr` directive you must specify `#address` when the register contains an address rather than an absolute numeric value.

Coding conventions

Storage size

The emulated real storage is the extent of the program loaded by `loadcore` console command. Do not store outside the extent of the program; reserve storage to be used for results by `DS` or `ORG` instructions.

Register usage

The following general registers contain a particular value, even after `sysclear`; it must be restored before returning.

11	Base register for *runtest* working storage.
14	Return address to *runtest*.
15	Base for real storage. You must use register 15 as a base rather than register 0. As register 15 will contain 0 when running native, all this costs is a register.

Returning to Hyperion

Loading a disabled PSW is not particularly useful on z/CMS; instead you must return by branch on general register 14.

You determine what to do by testing register 14 for zero.

```
 ltr 14,14     Have a return address?
 bnzr 14       return if so
 LPSWE WAITPSW     Load wait PSW
 ds 0d
WAITPSW dc x'0002000180000000',ad(0) OK wait state PSW
```

Restrictions

- Anything that alters processor architecture is out of bounds, such as `SIGP`, modifying the prefix register, loading a disabled wait PSW, *etc.*
- The machine is not all yours. Well, it *is*, but you may not wish to exercise that prerogative. You are likely to want CMS to remain after running the test case.
- The `*Gpr` directive to verify an address must add the comment `#address` so that the value can be rebased.

Appendix C. The tests directory

The `tests` directory holds test scripts of various types.

File extensions

While a UNIX file can have just about any name, the following file extensions are used by convention:

.asm	Source files for test cases that are designed for assembly by ASMA (Harold Grovesteen's SATK). `.core` files are produced directly by ASMA. Refer to Appendix A, "Using ASMA to assemble testing programs" on page 22.
.assemble	Source files for test cases that are designed for assembly with HLASM. Download the TEXT file (in binary, please) and use `text2tst.rexx` to generate the test case that is actually run. On VM/370, you can use the system assembler to assemble, but you will likely do the conversion on the workstation. `http://vm.marist.edu/~pipeline/zops.copy` contains a stacked copy file for many new operation codes.
.core	Binary core image files loaded into main storage by the Hercules `loadcore` console command. They can be created by many means; popular tools are ASMA for direct creation and `mkcore.rexx` when a TEXT deck has been produced by a System/360 or later assembler.
.list	An assembler listing file produced by ASMA.
.listing	An assembler listing file produced by HLASM.
.subtst	Secondary helper scripts that are called by a primary `.tst` script usually to run a script in all three architectures, where differences have been parameterised by setting Hyperion variables.
.tst	Files that are to be processed by the UNIX `runtest` command or the Windows `runtest.cmd` batch file. The output is subsequently inspected for correctness by `redtest.rexx`. You can create `.tst` files either "by hand" for use with `.core` files created by ASMA; or by assembly on a /360 system (VM, MVS, DOS), where TEXT decks are processed into `.tst` files by `text2tst.rexx`. `make check` gathers all `.tst` files, thus being the regression test for Hyperion.
.txt	Manually executed test scripts invoked via the Hercules `script` command. Their output must be manually eyeballed to ensure they executed correctly. Usually they end by loading a disabled wait PSW. If the PSW's instruction address is 0 then the test case likely completed successfully. These files are not part of the Hyperion regression suite.

Macro libraries

The subdirectory `tests/mac` contains macros for use with ASMA; they need the file name extension `.mac`.

Utilities

The `tests` directory contains a number of utilities that you may wish to upload to z/VM or download to your workstation.

mkcore.rexx—Convert TEXT deck to core image file

```
►►──mkcore.rexx──────────────────────────────────────────────────────►◄
```

The TEXT deck is read from standard input and the core image is written to standard output.

The core image is as large as the size of the first control section in the TEXT deck.

text2tst.rexx—Convert TEXT deck to test commands file

```
►►──text2tst.rexx────────────────────────────────────────────────────►◄
```

The TEXT deck is read from standard input and console command script is written to standard output.

Index

End of document