High Level Assembler and LD for Linux on System z

SHARE 107 Baltimore, MD
Session 8191 August 2006

David Bond
Tachyon Software LLC
dbond@tachyonsoft.com

ELF:
Executable and Linkable Format

Linux uses ELF files for:

- Object files
  created by assemblers
- Executable Programs
  created by linker
- Shared Objects
  created by linker
- Core Dumps
  created by Linux kernel, read by debugger (gdb)

ELF files are architecture-specific. For our purposes, there are two versions:

ELF32: Linux for S/390, 32-bit address words and lengths.

ELF64: Linux for System z, 64-bit address words and lengths.

HLASM can produce ELF32. The Tachyon z/Assembler can produce both ELF32 and ELF64.

Linux for System z contains a compatibility interface for 32-bit programs, allowing most Linux for S/390 programs to run.

Most (all?) Linux compilers (such as gcc) create assembler source code and invoke the assembler (as) to create object files.

ELF sections

Read-only code and data (.text)
- HLASM RSECT creates .text.RSECT section
- Linux/390 executable program starts at X'00400000' (4M)
- z/Linux executable program starts at X'80000000' (2G)

Read-only data (.rodata)
- Not created by HLASM
- Follows .text sections

Data with initial values (.data)
- HLASM CSECT creates .data.CSECT section
- Starts on next 4K page after read-only sections

Data with initial X'00' value (.bss)
- HLASM COM creates .bss.*name* section, where *name* is COM label
- Follows .data sections

Tachyon z/Assembler sections:

Assembler code		Section
_TEXT	RSECT	.text
_RODATA	RSECT	.rodata
_DATA	CSECT	.data
_BSS	COM	.bss
other	COM	named .bss areas

ld: the Linux Linker

Inputs:

- Object files (.o)
- Object file archives (.a)
- Shared Objects (.so)
- Command line options
- Linker script files

Outputs:

- Executable programs
- Shared Objects
- Link map

Linker script files are more powerful than the z/OS binder control statemenst.

Most linker script statements tell ld how to combine, align and order sections.

The default script file is usually used, but it is possible to replace all or parts.

To see the linker script, issue:
ld -verbose

Running ld

From the command line, program name: ld
All options are available, but no defaults beyond what is in the default script.

Via gcc
Much easier to invoke ld this way when including the glibc routines than to specify all of the options.
Override ld options using -Wl,*options*
See how ld is invoked using gcc -v

Via make
Best way to specify how to build parts of a system using compilers and linker.

The option to produce a link map file using ld is:
-Map *filename*

The option to produce a link map using gcc is:
-Wl,-Map,*filename*

For example, if the object file created by HLASM is named hlahello.o, it can be linked using:
gcc hlahello.o -o hlahello -Wl,-Map,hlahello.map

This will create the executable program named hlahello and a text file containing the link map named hlahello.map

Archives and Shared Objects

Archives (.a files) are collections of object (.o) files. An index of external symbols is built when the archive is created using the ar command. Referenced routines are statically linked into the executable program.

Shared objects (.so files) are loadable collections of object files created by ld using the -shared option. Referenced routines in the executable program are resolved to shared objects, which are loaded at run-time.

Both archives and shared objects can be input to the linker when creating an executable program. The glibc shared object is automatically included when ld is invoked via gcc.

When an executable program is loaded, control is given to the dynamic linker (ld.so) which loads all needed shared objects and resolves the references.

The dynamic linker looks for shared objects in:

1. The directories listed in the LD_LIBRARY_PATH environment variable.
2. The list of directories in /etc/ld.so.cache, which is maintained by ldconfig.
3. /lib
4. /usr/lib

HLASM code can reference code and data in archives and shared objects. HLASM code can be put into archives, but it is not easy to write assembler routines that will work in a shared object because they must be Position Independent Code. (See the gcc -fpic option.)

glibc Runtime Environment

Initialization and termination provided
- Shared objects are loaded
- R15 set as stack pointer
- Environment variables are set
- main is called with command-line parameters
- atexit routines are called when main returns

Access to C library functions
- Buffered I/O
- No need to know SVC numbers

Link with routines written in gcc-based languages

ld sets the “entry point” of an executable program to the label _start, which is normally provided by glibc.

Invoke kernel “write” service:

```
        LHI   R2,1          get stdout file handle number
        LA    R3,buffer     get address of data to write
        LHI   R4,L'buffer   get length of data to write
        SVC   4             invoke kernel write service
```

Invoke glibc “write” routine:

```
        LHI   R2,1          get stdout file handle number
        LA    R3,buffer     get address of data to write
        LHI   R4,L'buffer   get length of data to write
        L     R1,=V(write)  get address of glibc write routine
        BASR  R14,R1        call write routine
```

Linux Register Usage

R0-R1	Not preserved
R2-R3	Not preserved, parameters and return values
R4-R5	Not preserved, parameters
R6	Preserved, parameter
R7-R12	Preserved
R13	Preserved, often literal pool base register
R14	Not preserved, return address
R15	Preserved, stack pointer
F0	Not preserved, parameter and return value
F2	Not preserved, parameter
F4,F6	Preserved, z/Architecture parameters
F1,F3,F5,F7-F15	Not preserved
Access Registers	Not preserved, normally unused

Function return values:

Type	Linux for S/390	Linux for zSeries
char, short, int, long, * or 1, 2 and 4-byte structures	R2	R2
long long and 8-byte structures	R2 and R3	R2
float, double	F0	F0

Function parameter values:

The first 5 integer (char, short, int, long, long long) and pointer parameters are passed in registers R2, R3, R4, R5 and R6. For Linux for S/390, long long parameters are passed in register pairs. Structures of 1, 2, 4 or 8 bytes are passed as integers.

In Linux for S/390, the first 2 floating point parameters are passed in F0 and F2. In Linux for System z, the first 4 floating point parameters are passed in F0, F2, F4 and F6.

All other parameters are passed on the stack. If the return value is not an integer, pointer, float, double or 1, 2, 4 or 8-byte structure, a “hidden” parameter in R2 will contain the address of the return area.

Linux Stack Frame

S/390	System z	Purpose
0-3	0-8	back chain
4-15	8-31	reserved
16-23	32-47	scratch area
24-63	48-127	saved r6-r15 of caller function
64-79	128-143	saved f4 and f6 of caller function
80-95	144-159	undefined
96	160	outgoing args passed from caller to callee
96+x	160+x	possible stack alignment to 8-bytes
96+x+y	160+x+y	alloca space of caller (if used)
96+x+y+z	160+x+y+z	automatics of caller (if used)

Only the registers that are modified by a function need to be saved.
The stack grows toward lower addresses.

Stack frames are always double-word aligned.

The stack of the process's initial thread starts at the high end of virtual storage and
grows down. For Linux for S/390, the high end is X'7FFFFFFF' (2G-1).
In Linux for System z 2.6, 42 bits of the 64-bit possible virtual storage range are used, so the highest address is X'000003FF FFFFFFFF' (4T-1).

Linux threads are like z/OS tasks. Each thread has its own stack.

The Application Binary Interface (ABI) specification defines the register usage, stack frame format and other interface information.

The ABI specification for Linux for S/390:
http://www.linuxbase.org/spec/ELF/zSeries/index_s390.html

The ABI specification for Linux for System z:
http://www.linuxbase.org/spec/ELF/zSeries/index.html

Porting Issues from z/OS or VSE

Register contentions are different.
- R15 must be used as the stack pointer
- R13 is not used as a save-area pointer
- Cannot use BAKR/PR

Services are C-language based, not assembler.
- No DCB/DTF, ACB, CVT/COMREG, ...
- No LOAD, LINK, ATTACH, ...

File structures.
- Linux files are just bytes, no records or blocks.
- No VSAM.
- DB2 and other database engines are available.

Character Set issues are likely.

There is no 24-bit addressing mode, so very old programs can be more difficult to port to Linux. All Linux/390 addresses are 4 bytes.

The good news is that programs don't have to worry about storage “below the line” vs. “above the line”. The bad news is that 3-byte relocatable address constants cannot be assembled or linked on Linux.

Porting Strategy

- Isolate logic from file and operating system interfaces.
- Save R15 in a global variable.
- Switch to and use z/OS register conventions everywhere possible.
- Create interface routines to perform I/O and other services.
- Service routines need to switch back to Linux register conventions.

Remember to separate code from data.
.text sections cannot be modified.
It is easier to port reentrant modules.

Watch out for signals!

Good candidates for conversion to Linux are programs that already isolate operating system interfaces from logic.

Examples are programs that are designed to be easily ported between z/OS and VSE. One of the best examples is HLASM itself.

Signals are normally processed by calling the designated signal handler using the current stack frame address in R15. If R15 does not point into the stack, bad things will happen. The solution is to tell the kernel to use an alternate stack when setting the signal handler.

Sample “glue” module

```
main    RSECT
main    ALIAS C'main'
        STM   R6,R15,24(R15)        Save caller's registers
        BASR  R12,0                 Set base register
        USING *,R12
        LR    R1,R15                Get address of caller's frame
        AHI   R15,-96-72 -8         Allocate stack storage
        ST    R1,0(,R15)            Save address of caller's frame
        LA    R13,96(,R15)          Get address of save area
        L     R10,=V(STACKPT)       Get address of stack pointer
        ST    R15,0(,R10)           Save stack pointer
        LA    R1,72(,R13)           Get address of parm. list
        STM   R2,R3,0(R1)           Save argc and argv
```

This (incomplete) routine is intended to be used when the top-level routine is an assembler module that was ported from z.OS or VSE.

After saving the caller's registers on the stack and setting up the stack frame, it sets R13 to point to a standard 72-byte format-1 save area. It then saves the address of the current stack frame into the global data area named STACKPT. Subsequent modules that need to restore the stack frame address before calling a C-language module will get the value from STACKPT.

To prepare for call to the top-level traditional assembler routine (named START), the routine sets R1 to point to a 2-word parameter list. The first parameter list word contains the number of command line parameters. The second word contains the address of an array of addresses of the parameters.

Each parameter is a NUL-terminated ASCII string. The first string is the name of the program. The first parameter is the second string, and so on.

Sample “glue” module

```
         LR      R11,R15                 Save stack pointer
         L       R15,=V(START)           Get address of first module
         BASR    R14,R15                 Call first module
         LR      R2,R15                  Get return value
         LM      R6,R15,96+72+8+24(R11)  Restore caller's registers
         BR      R14                     Exit program
         LTORG

STACKPT  COM
         DS      A                       Stack pointer
```

The stack frame address is saved in a non-volatile register so the caller's registers can be quickly restored. Then the START module is called using standard z/OS or VSE linkage:
R1 contains the address of a parameter list
R13 contains the address of a 72-byte save area
R14 contains the return address
R15 contains the called module's address

Upon exit, the return value in R15 is transferred R2, the Linux return register. The the calling routine's registers are restored from the stack, including the caller's stack frame address and the return address; and control is returned to the caller.

The caller is the glibc initialization routine. When control is returned from main, the exit() library routine will be called.

Note: This code does not preserve the floating point registers.

Sample service module

```
CALLPUTS    RSECT
            STM     R14,R12,12(R13)     Save caller's registers
            LR      R12,R15             Load base register
            USING   CALLPUTS,R12
            L       R15,=V(STACKPT)     Get address of stack pointer
            L       R15,0(,R15)         Restore stack pointer
            L       R2,0(,R1)           Get address of string to put
            L       R1,=V(puts)         Get address of puts routine
            BASR    R14,R1              Call puts()
            LR      R15,R2              Get return value
            L       R14,12(,R13)        Restore return address
            LM      R0,R12,20(R13)      Restore caller's registers
            BR      R14                 Return to caller
puts        ALIAS   C'puts'
```

This illustrates how to write a wrapper function for the C library routine “puts” which takes one parameter, the address of the null-terminated string to be written to stdout. The wrapper function is named CALLPUTS.

CALLPUTS using z/OS standard linkage conventions:

R1 contains the address of a one-word parameter list

R13 contains the address of a 72-byte format-1 save area

R14 contains the return address

R15 contains the address of CALLPUTS on entry and the return value on exit.

The “puts” routine must be called with Linux/390 linkage conventions:

R2 contains the address of the string on entry and the return value on exit

R14 contains the return address

R15 contains the caller's stack frame address

Character Set Handling

Ported programs will often need to handle both EBCDIC and ASCII or Unicode.

Data files may remain encoded with EBCDIC characters.
Parameters will be ASCII.
Text files will likely be ASCII for ease of viewing and editing.
Messages will need to be produced in ASCII.

HLASM 1.5 supports CA, CE and CU types for DC and literal values.
TRANSLATE(AS) option sets default for C-type values to CA.

HLASM 1.5 does not support CA and CU for self-defining terms.

If most constants are ASCII, set TRANSLATE(AS) and use CE where needed.
If most constants are EBCDIC, set NOTRANSLATE and use CA where needed.

HLASM supports:

```
CLC =CA'x',STRING
```

but not:

```
CLI STRING,CA'x'
```

The Tachyon z/Assembler supports CA, CE and CU self-defining terms. This assembler also supports the TRANSLATE(AS,SELF) option to change the default for self-defining terms to be CA.

ED and EMDK instructions produce EBCDIC digits, so the result needs to be translated to ASCII.

Web Resources

High Level Assembler:
http://www.ibm.com/software/awdtools/hlasm/

Tachyon z/Assembler:
http://www.tachyonsoft.com

ELF and DWARF for S/390 Links:
http://www.tachyonsoft.com/elf.html

ABI Specifications:
http://www.linuxbase.org/spec/ELF/zSeries/index_s390.html
http://www.linuxbase.org/spec/ELF/zSeries/index.html

GCC:
http://gcc.gnu.org/

binutils (including ld):
http://www.gnu.org/software/binutils/