An Assembler Programmer's view of Linux for S/390 and zSeries

SHARE 99 San Francisco, CA
Session 8139 August 2002

David Bond
Tachyon Software LLC
dbond@tachyonsoft.com

Why would anyone care about assembler on Linux/390?

Unlike z/OS where the system interface is defined by assembler macros, Linux programs can be written entirely in C. In Linux you don't need to code in assembler just to get something done.

So why does assembler matter on Linux for S/390 and zSeries?

- Understanding how Linux/390 really works and how to debug when things can go wrong.
- Coding for extreme performance.
- Porting assembler code from another S/390 Operating System.
- Using GCC for Linux/390 to build programs for z/OS.

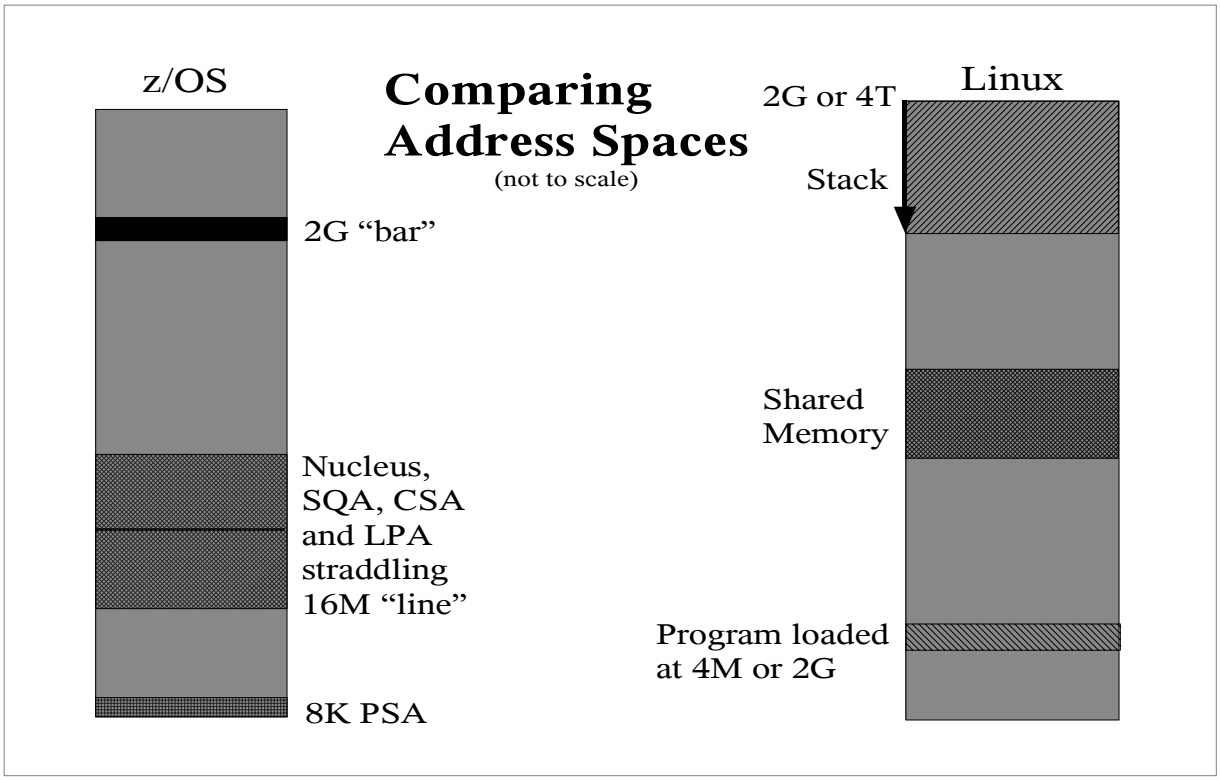
Linux Address Spaces

Like z/OS, Linux is a multi-address space operating system. Each Linux process runs in its own address space. (Processes can also have multiple "threads" which correspond closely to MVS subtasks.)

In z/OS, most system service requests do not require an address space switch. Most service requests (OPEN, READ ...) result in a switch to supervisor state and the calling task executes "nucleus" code that is mapped into all address spaces.

In Linux, the "kernel" code is only in the Kernel Address Space ("init" process) so all system call requests cause a space switch. However, the kernel code is executed under the original thread so, like z/OS, no task switch is needed.

All structures (a.k.a. Control Blocks) used to manage processes, threads and their resources are in the Kernel Address Space. Access to these structures is only available through certain system calls. The /proc/ file system can also be used to gain access to system information.



In Linux, the Kernel Address Space has a different layout from all other address spaces which are depicted above.

The Prefixed Storage Area (PSA) is only mapped into the Linux Kernel Address Space, not normal address spaces. Location 0 is normally an invalid virtual storage address because an address value of 0 is used in the C language to indicate an invalid pointer value. In Linux, location 0 cannot be referenced or modified without causing a Translation Exception (SIGSEV signal).

Because the PSA is mapped into all z/OS address spaces, z/OS can only trap attempts to modify location 0. z/OS cannot trap references to low memory.

Kernel and User Modes

The Kernel Address Space is always the Primary Address Space. User processes run in Home Space ASC mode unless they are invoking a kernel service.

A system call (SVC) switches from User Mode (Home Space, Problem State) to Kernel Mode (Primary Space, Supervisor State). Once in the kernel code, parameters are fetched and stored from/to the calling address space using Secondary Space or AR ASC mode.

In z/OS terms, Linux processes run in PASN≠SASN=HASN.

The Home Space Switch Event bit is set in Control Register 13. This will cause a Program Check Interrupt for a Space Switch Event should a user-mode program "accidentally" try to switch to Primary Space or Access Register ASC mode using the SAC instruction.

In S/390 and z/Architecture, a program can have access to up to 3 different address spaces. These are called: Primary, Home and Secondary. Each corresponds to a different control register containing the address translation control information, including the segment/region table address.

In MVS, programs start out with the Primary, Home and Secondary address spaces all set to the job step address space (PASN=SASN=HASN). The Home address space is never changed and always identifies the address space of the current unit of work (TCB). The program can change its Secondary address space if it has authority to do so. The Primary address space is changed by a space switching PC and restored by the corresponding PR or PT instruction. A program can change which of these address spaces are being referenced using the SAC or SACF instructions.

The fourth Address Space Control (ASC) mode is called Access Register (AR) mode, which enables the use of the access registers to reference data in other address spaces and data spaces. In AR mode, instructions are fetched from the Primary Address Space.

PSW and Control Registers

These are example values from a Linux/390 user process:

```
PSW:  0709C000 804005E8
CR0:  14B52A02   CR1:  0021207F
CR2:  00000000   CR3:  00000000
CR4:  00000000   CR5:  00000000
CR6:  10000000   CR7:  8625817F
CR8:  00000000   CR9:  00000000
CR10: 00000000   CR11: 00000000
CR12: 00000000   CR13: 8625817F
CR14: D0000000   CR15: 00000000
```

PSW is Enabled, Problem State, Key Zero, Home Space.

CR0 is set for Low Address Protection, No Extraction Authority, Secondary Space Control, No Address Space Function.

CR7 and CR13 (Secondary and Home Translation) are set for Space Switch Event, and Private Space.

Linux does not use much of the complexity of ESA/390 and z/Architecture.

In Linux, there is/are no:

- Storage keys
- Linkage Stack
- ASN Tables
- ASTE
- Linkage Table
- Trace Table

Recent releases of the Linux kernel do set up the DUCT address in CR2.

Control Instructions

Most of the Control Instructions described in *Principles of Operation* cannot be used in User Mode because they must be executed in supervisor state or because:

BAKR, EREG, ESTA, MSTA, PC, PR:
No linkage stack

EPAR,ESAR, IAC, IPK, IVSK:
Extraction Authority bit in CR0 is zero.

MVCP, MVCS, MVCDK, MVCK, MVCSK, SPKA:
PSW Key Mask is all zero.

PT, SASN:
ASN Translation Control bit in CR14 is zero.

SAC, SACF:
Home Space Switch Event in CR13 is set.

Unless you are writing code that runs in the Kernel Address Space, such as a device driver, there is really no opportunity to use the Control Instructions in Linux. Writing such code is beyond the scope of this discussion.

If you want to learn more about writing kernel code, remember that Linux is open-source. The source is usually included in your Linux distribution and is available on the Internet at ftp://ftp.kernel.org/pub/linux/kernel/ or one of the many mirror sites.

General Instructions

The instructions described in *Principles of Operation* as General Instructions, Decimal Instructions and Floating Point Instructions may all be used in normal programming on Linux for S/390 and zSeries. Linux requires the Relative and Immediate instructions to be supported by the hardware so you can assume these instructions are available when you write Linux code.

GCC and the C library have no direct support for packed decimal data and the Decimal Instructions. Nothing in Linux prevents you from writing assembler code to work with packed decimal data.

Linux can provide emulation for the Binary Floating Point instructions if they are not available in the S/390 hardware. The Hexadecimal Floating Point instructions can be used, however GCC and the C library functions only support the Binary Floating Point (IEEE) formats.

The good news is that your knowledge of the S/390 machine instructions transfers over to programming Linux/390 in assembler.

The bad news is that the linkage conventions, system calls and memory management are completely different on Linux than any other IBM S/390 operating system.

Linux System Calls

The SVC instruction is used to invoke Linux System Calls. The SVC instruction switches to supervisor state in Primary Space ASC mode, switching to the Kernel Address Space.

The linkage convention is identical for all system calls and is essentially the C language linkage as defined for Linux:
0 to 5 parameters are passed in registers 2 through 6, starting with register 2. The return value is in register 2. The thread's stack in the user address space is not used. Instead, the system call code uses a per-thread stack area in the Kernel Address Space.

Sample System Call:

```
LA    2,0          # File handle 0
LA    3,buffer     # Where to read byte
LA    4,1          # Bytes to read
SVC   3            # Read
```

number of characters read (0 or 1) or error indication is returned in R2.

If an error occurs, R2 will contain a value between -1 and -122, which correspond to the errno values 1 to 122.

You can see a list of the SVC numbers corresponding to the various system calls in the file /usr/src/linux/include/asm-s390/unistd.h

Linux Register Usage

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

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 zSeries, 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	zSeries	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 zSeries 2.4, 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 OS/390 tasks. Each thread has its own stack.

Tool Time

Now that you know everything about the instructions, system calls and linkage conventions for assembler coding for Linux for S/390 and zSeries, you need to know about the tools available for Linux.

Every assembler programmer needs:

- Assembler
- Linker
- Debugger
- Libraries
- Editor

Good programmers need source configuration and management tools too! Linux provides several good source library and version management tools.

The most valuable configuration tool is “make”. Once you define the parts of a program or system to “make” and how to build some parts (object, executables) from other parts (source), it is very easy to build a program or system.

Terminology

z/OS	Linux
C/C++ Compiler	gcc
HLASM	as
Binder	ld
TSO TEST, IPCS	gdb
GOFF object file	ELF relocatable
Program Object	ELF executable
DLL	ELF Shared Object
Dump Data Set	ELF “core” file
SYM records, ADATA	DWARF
Language Environment	glibc

ELF Executable and Linkable Format

Format of object, executable and dump files

DWARF Debug With Arbitrary Record Format

Format of debugging records inside ELF files

GCC GNU Compiler Collection

C, C++, JAVA, FORTRAN, ADA and other compilers

GAS GNU Assembler

Assembler included with GNU tools

Linux Development Packages

gcc	GNU Compiler Collection (C, C++, Objective C, FORTRAN, JAVA,ADA)
binutils	assembler, linker
gdb	debugger
glibc	GNU C/C++ library
make	make

Each of these packages are available at different levels. A given Linux distribution will provide a copy of these packages at some level. You can install other levels, as long as you watch for compatibility. For example, a given level of gcc may require a certain minimum binutils level.

For example, binutils 2.10 is required for gcc 3.0, at least for the S/390 version. gcc 3.0 generates some instruction operation codes that were not supported by the GNU assembler prior to binutils 2.10.

In Linux terminology, “s390” is for the 31-bit version (Linux for S/390) and “s390x” is for the 64-bit version (Linux for zSeries). GCC supports both s390 and s390x from the same program. The default is s390 (31-bit). Use the -m64 compiler option to generate s390x (64-bit z/Architecture) code.

binutils and glibc must be built for either s390 or s390x and the correct version must be used.

Compiling Assembler Code

You have 3 ways of compiling assembler code for Linux:

1. **GNU assembler** (GAS) - no macros, DSECTs or USINGs, different syntax than traditional HLASM.
2. **Assembler instructions embedded in C or C++** GCC has an arcane syntax for embedding assembler instructions in-line in C/C++ code. This is nice for inserting small amounts of assembly code in a C/C++ program when needed. The syntax is even worse than GAS.
3. **Tachyon z/Assembler** - can read HLASM or GAS syntax and produce any object format, including ELF for Linux. Legacy assembler code, complete with macros and DSECTs can be moved to Linux. GAS code produced by GCC can be compiled for OS/390 or z/OS targets.

GCC produces GAS code, normally in a temporary file, before invoking the assembler to produce object code. Embedding assembler statements in C/C++ code causes those statements to be generated in the file sent to the assembler.

You can tell GCC to avoid invoking the assembler and instead to keep the assembler source file if you want to examine the generated assembler code.

Debugging Linux Programs

The GNU debugger (gdb) can be used to debug Linux programs in any language. gdb can be used to debug programs while they are running. gdb can also be used to examine "core" files produced when a process is terminated by certain signals - e.g. SIGSEGV (S0C4 in MVS terms).

Unless the symbolic debugging information is stripped from an executable module, gdb can match the memory of an executing program or core file to external symbols, allowing symbolic debugging. The -g option of the GCC compiler generates even more debugging information, allowing source-level debugging.

If you do not see core files being created, issue the **`ulimit -a`** command. It will probably show that the core file limit is 0. This can be changed by issuing the **`ulimit -c unlimited`** command.

The elfdump program can format the contents of S/390 and Intel/x86 ELF files, including core files, object files, executable programs and shared objects. The elfdump program is available via http://www.tachyonsoft.com/elf.html

The GNU Assembler

The GNU Assembler (GAS) is provided with your Linux distribution in the binutils package. If you intend to code in assembler using GAS or examine the output of the GCC compiler, you need to understand GAS syntax and how it differs from the traditional HLASM syntax.

GAS syntax is based on a number of non-mainframe assemblers, not HLASM, which makes it foreign to most mainframe assembler programmers. Fortunately, the machine instruction syntax is similar between HLASM and GAS.

GAS Syntax

GAS is free-form. Continuation is indicated by ending a line with a backslash (\), just like C. Multiple statements are allowed per line, each ended with a semicolon (;).

Comments start with a pound sign/hash mark (#) and continue for the rest of the line. Continuation is allowed.

Labels end with a colon (:). Assembler directives start with a period (.). Everything else is a machine instruction.

Symbols consist of the letters A-Z, a-z, digits 0-9, underscore (_), dollar sign ($) and period(.). Symbols are case sensitive.

GAS has no HLASM-like macro facility.

UNIX and LINUX are C-based, so it is natural that the GNU assembler syntax details are very C-like.

GCC allows assembler instructions to be included in C and C++ code using the asm language extension. GCC writes these assembler instructions directly to the GAS file.

The primary use of GAS is as a compiler back-end. Very little of LINUX code is written in assembler, even in the kernel. Because few humans code in GNU assembler, GAS does not need the level of sophistication found in HLASM (powerful macros, DSECTs, USINGs, good diagnostics, etc.)

Registers, Labels, Numbers

Registers:

`%r0 - %r15`	General registers
`%f0 - %f15`	Floating point registers
`.`	Current location counter (like HLASM `*`)

Labels:

.Lxxxx:	Internal label – not visible in debugger
`0: - 9:`	temporary labels, referenced by (for example) `0B` (backward reference) or `1F` (forward reference).

If not defined, symbols are assumed to be external references.

Numbers:
C-like syntax, if starts with 0x, remainder of number is hexadecimal, if starts with 0b, remainder of number is binary, otherwise if starts with 0, number is octal, otherwise decimal.

Temporary labels are very nice features of the GNU assembler language. They allow local labels to be defined without the need to resort to HLASM's &SYNDX variable symbol. For instance, a reference to **`2F`** is resolved to the next (forward) definition of the temporary label **`2:`** and a reference to **`9B`** is resolved to the nearest previous (backward) definition of the temporary label **`9:`**.

As an example, the following code is generated by GCC for the strlen() function:

```
   sr    0,0
   lr    %r12,%r11
0: srst  0,%r12
   jo    0b
   lr    %r12,0
   sr    %r12,%r11
```

This same code sequence could be generated later without any conflict between the definitions of **`0:`**.

Machine Instructions

GAS machine instructions are similar to HLASM format, except:

In RX instructions, the base and index registers are reversed.
If only one register is specified, it is assumed to be a base register.

Additional branch mnemonics (GAS: HLASM)

JHE:	BRC 10	JLE:	BRC 12	JLH:	BRC 6
JNHE:	BRC 5	JNLE:	BRC 3	JNLH:	BRC 9

Same for BC and BCR mnemonics.

Different BRCL mnemonics (GAS: HLASM)

JG:	JLU	JGE:	JLE	JGH:	JLH
JGHE:	BRCL 10	JGL:	JLL	JGLE:	BRCL 12
JGLH:	BRCL 6	JGM:	JLM	JGNE:	JLNE
JGNH:	JLNH	JGNHE:	BRCL 5	JGNL:	JLNL

... etc.

GCC/GAS introduced extended mnemonics for the 6 condition code masks missing from the HLASM extended mnemonics. Unfortunately, the GCC/GAS JLE and JLH extended mnemonics for BRC conflict with the HLASM extended mnemonics for BRCL. GCC/GAS replaced the JLxxx (Jump Long) extended mnemonics for BRCL with the JGxxx (Jump Grande) mnemonics.

See http://www.tachyonsoft.com/txac.htm for instruction tables that include the GCC/GAS extended mnemonics. These tables include instructions from all of the Principles of Operation books as well as instructions published in other IBM manuals or discovered from other sources.

Storage Definition Directives

GAS	HLASM
.byte	DC X'xx'
.short	DC HL2'nn'
.long	DC FL4'nn' or DC AL4(symbol)
.quad	DC FDL8'nn' or DC ADL8(symbol)
.single	DC EBL4'nn'
.double	DC DBL8'nn'
.comm	COM, DS
.ascii	DC C'ccccc'
.string	DC C'ccccc',X'0'

GAS does not align constants based on type.
GCC does not produce .single and .double – IEEE value is generated by .long 0xXXXXXXXX constants.
.ascii and .string use C-like syntax for characters, including escape values. (e.g. `"\n\0"`)

GAS does not align anything. Even machine instructions are not halfword aligned! However the .align directive can align to any power of 2.

.ascii and .string operands are strings enclosed in double quotes. The .string directive includes a terminating X'00' byte in the string. `.string "hello"` is the same as `.ascii "hello\0"`.

.ascii and .string operands can include “escape” values:
\a = X'07' (BEL) \b = X'08' (BS) \f = X'0C' (FF)
\n = X'0A' (LF) \r = X'0D' (CR) \t = X'09' (TAB)
\v = X'0B' (VT) \" = double quote \\ = back slash
\x<hex digits> = X'<hex digits>' \<octal digits>

.byte, .short, .long and .quad operands can be expressions. Numbers are decimal unless they start with a zero. Zero is the “escape” character, allowing octal, hexadecimal (0x) or binary values (0b). Character values can also be included (e.g. 'a') and support the same set of escape values as can be specified in .ascii and .string operands.

Miscellaneous Directives

GAS	HLASM
.align	CNOP
.file	none
.globl	ENTRY
.ident	none
.local	none
.org	ORG
.set	EQU
.size	none
.stab	ADATA
.type	XATTR
.version	none
.weak	WXTRN

Notes: .org is forward only. .align fills with X'07' bytes by default.

Unlike HLASM's EQU, .set can be used more than once for the same symbol, allowing a symbol to have a different value for different parts of the assembly. .set can also be used to change the value of the location counter in a forward direction: `.set .,.+4`

.align fills with X'07' bytes generating “NOPR 7” instructions in any halfwords.

The inability of the GNU assembler to support ORG to a previous location vastly simplifies the assembler logic!

Section Definitions

GAS	HLASM
.text	RSECT
.data	CSECT
.bss	COM
.section	RSECT/CSECT/LOCTR

GAS ELF sections are similar to HLASM GOFF classes.

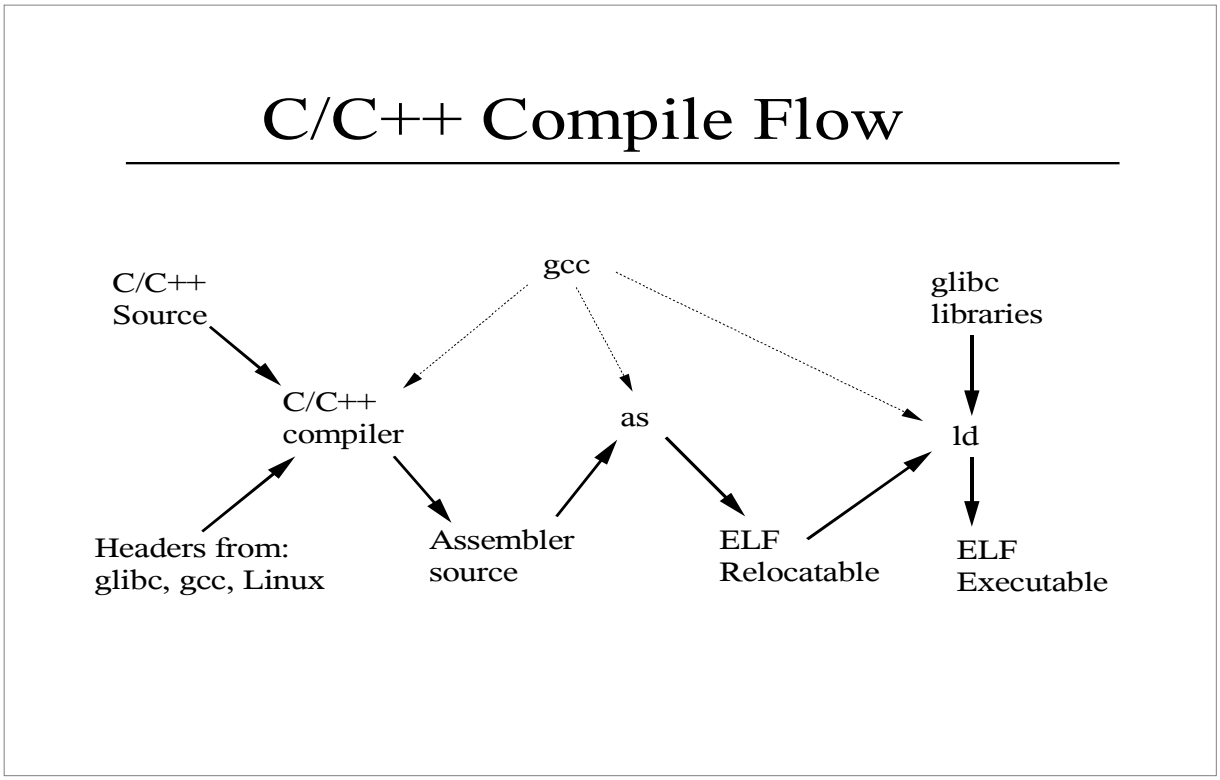
There is no GAS equivalent to DXD or DSECT.

.text is read-only executable code and literals
.data is modifiable initialized storage
.bss is modifiable uninitialized storage

Other sections may be defined, e.g. .rodata is for read-only data.

.text, .data, .bss and .rodata are the only sections generated by GCC for normal C code.

For C++ code, GCC generates many additional sections that are recognized by the GNU linker for special processing. These special sections are to handle static constructors and destructors and other C++ features that must be processed at link time.



gcc is the controller program for the compile process. Unless told to do otherwise, it will invoke the C/C++ preprocessor, C/C++ compiler, the assembler (as) and the linker (ld).

The C/C++ preprocessor reads the source and header files. The header files are provided by glibc, gcc, Linux and the application. The preprocessed C/C++ source code is passed to the C/C++ compiler.

The C/C++ compiler reads the preprocessed source code and generates an intermediate work file containing assembler source code (gas format). gcc can be told (via the -S option) to retain the assembler source file (filename.s) and to not invoke the assembler.

gcc normally invokes the GNU assembler and deletes the temporary assembler source file. The assembler produces an ELF relocatable object file, normally as a temporary work file that is passed to the linker. If gcc is invoked with the -c option, the object file (filename.o) will be retained and the linker will not be invoked.

If run without either -S or -c, gcc then invokes the linker and deletes the temporary object file. The linker combines the object file with object modules from various libraries (mostly glibc) and produces and ELF executable program.

You can run gcc with the -v option to see what programs are invoked.

C Source Program

```
>cat hello.c
#include <stdio.h>
int main(
     int argc,
     char * argv[])
{
     puts("Hello world.");
     return 0;
}
>gcc -O3 -S hello.c
```

This is the classic “Hello world” program in C.

The UNIX/Linux “cat” command can be used to print a file to the terminal.

The hello.c program is compiled with the -O3 flag for maximum optimization and the -S flag to cause the compiler to generate the assembler source file and stop. If the compile is successful, the assembler source will be put in a file named hello.s

GAS Source – page 1 of 2

```
>cat hello.s
   .file   "hello.c"          #data for .note section
   .section.rodata            #start .rodata section
   .align  2                  #halfword alignment
.LC18:                        #internal label
   .string "Hello world."     #define string constant
.text                         #start .text section
   .align  4                  #fullword alignment
.globl main                   #entry point definition
   .type   main,@function
main:
   stm   %r13,%r15,52(%r15)   #save caller's registers
   bras  %r13,.LTN0_0         #R13=literal base
.LT0_0:
.LC19:
   .long   .LC18              #address of string constant
```

This is the 31-bit GNU assembler code produced by GCC 3.02 for Linux for S/390. The comments generated by GCC are omitted. The comments to the right of the GNU assembler code were added by hand.

GAS Source – page 2 of 2

```
.LC20:
   .long   puts                      #address of puts function
.LTN0_0:
   lr    %r1,%r15                    #stack frame setup
   ahi   %r15,-96
   st    %r1,0(%r15)
   l     %r3,.LC20-.LT0_0(%r13)      #call puts function
   l     %r2,.LC19-.LT0_0(%r13)
   basr  %r14,%r3
   lhi   %r2,0                       #set return value
   l     %r4,152(%r15)               #restore return address
   lm    %r13,%r15,148(%r15)         #restore caller's registers
   br    %r4                         #return to caller
.Lfe1:
   .size   main,.Lfe1-main
   .ident  "GCC: (GNU) 3.0.2"        #data for .comment section
```

GCC Output – Notes

```
   .section .rodata
   .align  2
.LC18:
   .string "Hello world."
```

A half-word aligned, null-terminated ASCII string constant is defined in the .rodata section.

```
.text
   .align  4
.globl main
   .type main,@function
main:
   stm %r13,%r15,52(%r15)
```

An externally visible function named “main” is defined in the .text section. It is fullword aligned. The function saves only r13, r14 and r15 since those are the only non-volatile registers modified.

GAS uses the strongest .align directive within a section to set the alignment for the section. When combining sections, the linker uses the strongest alignment contributed by any object module for the section alignment within the program.

GCC generates the minimum STM instruction required to save registers that must be preserved when control is returned to the caller. When generating code, GCC first tries to use the volatile registers (0-5), which are the registers that a caller does not expect to be preserved across a call. If all of the volatile registers are used, GCC uses the highest numbered register available. This allows GCC to generate minimal STM/LM instructions in the function prolog and epilog.

GCC Output – Notes

```
   bras %r13,.LTN0_0
.LT0_0:
.LC19:
   .long   .LC18
.LC20:
   .long   puts
.LTN0_0:
```

Register 13 is set up as the base register for the literal pool. Two address constants are in the literal pool: the address of the string constant in .rodata and the address of the external function named “puts”.

```
   lr   %r1,%r15
   ahi  %r15,-96
   st   %r1,0(%r15)
```

A new stack frame of 96 bytes is allocated and the address of the caller's stack frame is saved (back chain).

If a function does not call another function, it is called a “leaf” function. Leaf functions that do not modify any of the caller's non-volatile registers do not need to save and restore any registers, so no stack frame is needed.

The GCC “literal pool” is actually a set of constants generated near the start of each function and usually addressed via R13 which is set up via the BRAS instruction. The label LT0_0 for this function is used as the base address of the literal pool when literals are referenced.

If local variables need to be allocated in the function's stack frame, more than 96 bytes would be needed and the AHI instruction would reflect this. The stack frame size is always a multiple of 8 to ensure that stack frames are doubleword aligned.

GCC Output – Notes

```
l     %r3,.LC20-.LT0_0(%r13)
l     %r2,.LC19-.LT0_0(%r13)
basr  %r14,%r3
```

Register 3 is loaded with the address of the "puts" function; register 2 is loaded with the first parameter; " puts" function called.

```
lhi   %r2,0
l     %r4,152(%r15)
lm    %r13,%r15,148(%r15)
br    %r4
```

The return value is loaded into register 2; the return address is loaded into register 4; registers 13 and 15 are restored; control is returned to the caller.

GCC generates explicit displacement values for literal references by subtracting the base address of the literal pool (LT0_0 in this case) from the address of the literal.

Unlike OS/360 standard linkage conventions, Linux/390 does not require that the called routine's address be in any specific register. Called routines cannot expect that the starting address is in a register.

In the call to the "puts" function, the load for the address of "puts" is performed as far in advance of the BASR as possible to reduce pipeline stalls. The load instructions for R2 and R3 can run in parallel.

In the function epilog, the three instruction sequence (L,LM,BR) is used instead of the minimal two instruction sequence (LM 13,15,148(15);BR 14) to allow the maximum amount of parallelism. In the three instruction sequence, the LM instruction can execute in parallel with the BR and subsequent instructions. See the proceedings of SHARE 98, session 8158 for a discussion of these performance issues.

Because GCC understands how to exploit instruction parallelism, it generates code that is unusual (and faster) than most human-coded assembler. Human-generated assembler should be maintainable, so it is not good practice to separate related instructions in non-obvious ways.

How good is GCC?

GCC is open-source, so each port is as good as the effort put into it.

- Intel/x86 optimization is very good.
- Dave Pitts' I370 port generates HLASM and works with OS/390 USS and LE, however the optimizer is broken.
- S/390 and S/390x (z/Architecture) ports are excellent. They were contributed by the same IBM group that writes millicode for the Z900. They have a deep understanding of the S/390 and z/Architecture, so the optimizer is outstanding (and getting better).

Major improvements as of gcc 3.02 and 3.1 – try it!

Dave Pitts' I370 port is available in source and executable form from:
http://www.cozx.com/~dpitts/gcc.html

The GCC home page is:
http://www.gnu.org/software/gcc/index.html
The GCC site includes information about recent and upcoming releases and links to sites from which the source can be downloaded. It is relatively easy to download, compile and install a new release of GCC. Patch files can be downloaded to upgrade the source of one release to another. (e.g. From 3.0.2 to 3.0.3)

At this time, GCC 3.11 is current and 3.20 may be available by the time this information is presented.

GCC releases are controlled by the GCC Steering Committee. Resources are largely provided by Cygnus/Red Hat.

The primary maintainer of the Linux for S/390 and zSeries version of GCC is Hartmut Penner of IBM Germany.

GCC for z/OS?

Why can't GCC be used for z/OS?

- GCC assumes ASCII.
- GCC needs GLIBC. GLIBC is for UNIX/Linux and ASCII.
- GCC generates GNU assembler code. GAS generates ELF.
- GCC debugging information is DWARF, which is not (yet) supported by any z/OS debuggers.
- z/OS Program Objects must be created by the binder. The binder does not read ELF.
- C++ ELF object modules must be processed by the GNU linker. The GNU linker reads and writes ELF. The z/OS loader cannot load ELF load modules.

One solution: Dave Pitts altered GCC to accept EBCDIC, produce HLASM and interface with LE.

Another solution would be to provide an ELF loader for z/OS. This would allow all of the GNU tools to be used: GCC, GAS, GNU linker and most of GLIBC. The ASCII problem would still need to be solved.

The LIB390 solution: assemble the GNU assembler code produced by GCC directly to GOFF and provide a replacement for GLIBC.

GCC for z/OS!

GAS/ELF Problem

- GAS syntax assembler can be converted to HLASM syntax for GOFF. The Tachyon z/Assembler can automatically assemble GAS source and create GOFF.
- GOFF can be linked by the z/OS binder to create normal z/OS load modules.

ASCII Problem

- For now, use patches from Dave Pitts' I370 port or use the ASCII<->EBCDIC translation support in LIB390.
- Integrated GCC support for EBCDIC from Cygnus/Red Hat is in beta test and should be released soon.

The Tachyon z/Assembler can be used for free when used to assemble the output of the GCC compiler. The free version can generate either GOFF (31-bit) or ELF (31-bit or 64-bit) object files from GNU assembler source. The free version of the Tachyon z/Assembler can be downloaded from the Tachyon Software web site at http://www.tachyonsoft.com

By assembling GAS to GOFF, no changes to GCC are required. This takes advantage of all of the work done by the GCC maintainers, including the excellent zSeries instruction optimizer.

In 64-bit code generated by GCC, the operands of the BRASL, BRCL and LARL instructions are often external references which can be resolved by the GNU linker. Unlike ELF, GOFF does not have defined relocation types for the operands of these instructions, so there is no way to assemble the 64-bit code generated by GCC into a GOFF object file. Until IBM adds support in GOFF and the binder, this problem can be fixed using a prelinker.

GCC for z/OS!

GLIBC Problem

- Tachyon Software has started an open-source version of GLIBC for 31-bit OS/390 and z/OS, called LIB390.
- LIB390 is LGPL code, so it cannot be statically linked with non-open-source products. A goal of the project is to make LIB390 into a "DLL" so that GCC and LIB390 can be used in commercial products.

C++ Problem

- C++ support will require a prelinker to perform the "magic" currently performed by the GNU linker for C++.

The source and object files of LIB390 can be downloaded from the Tachyon Software web site at http://www.tachyonsoft.com/lib390.html

LIB390 is based on GLIBC with changes and replacement routines as required for OS/390 and z/OS. Since GLIBC is distributed under the GNU Library General Public License (LGPL), LIB390 is also distributed under the same license. One provision of the LGPL is that any user of a program linked with GLIBC must be allowed to replace the GLIBC routines, usually by relinking. This usually requires the program to be distributed in object module form so the user can relink it, or else the program should be dynamically linked to GLIBC. It is intended that a future version of LIB390 can be dynamically linked with programs, allowing commercial products to be built with LIB390 while complying with the LGPL.

LIB390 is also being ported to MVS 3.8, the last open-source version of MVS. This will allow GCC to build programs for MVS 3.8. Work is underway to extend MVS 3.8 to support binary floating point. One goal is to then use GCC to port the Linux TCP/IP stack to MVS 3.8.

GCC for z/OS!

Debugger Problem

A DWARF-based debugger is needed. Anyone want to contribute?

GCC and GAS do not run on z/OS

- For now, GCC can be run on Linux/390 or as a cross-compiler on Windows or Linux/x86. Linux/390 can be run on Windows or Linux/x86 under Hercules. The Tachyon z/Assembler runs on Linux/390, Linux/x86, Windows, AIX and Solaris.
- A project goal is to allow GCC to build itself to run on z/OS.

Using cross-platform development tools like GCC and the Tachyon z/Assembler, you can build programs on one platform to be executed on another. For instance, you can build the object files for a program on Linux and then upload them to z/OS where they can be bound into Program Objects or Load Modules and executed.

With the limited TSO access on z/OS.e, cross-platform development is probably the preferred method. The only other choice would seem to be a telnet session into z/OS Unix System Services.

Since IBM FORTRAN and COBOL programs cannot be run under z/OS.e, GCC FORTRAN (g77) and GNU COBOL could be used to build programs for z/OS.e since LE would not be needed. The GNU COBOL home page is: http://www.gnu.org/software/cobol/cobol.html

Hercules is an open-source System/370, System/390 and z/Architecture emulator. Using Hercules, you can have the fun of installing and running Linux for S/390 and zSeries on your PC, Macintosh or whatever!
You can download Hercules from: http://www.conmicro.cx/hercules
Hercules will be discussed in session 2880 (6:00PM Tuesday).

How Can You Help?

Cygnus/Red Hat is providing EBCDIC support in GCC.

Tachyon Software is contributing a free version of the Tachyon z/Assembler for GCC and the runtime library support.

Your help is needed:

- Runtime library
- C++, COBOL and FORTRAN support
- Debugger and Profiler
- VSE and CMS support

The goal is to have a common set of open-source compilers and other development tools across all IBM mainframe operating systems: z/OS, OS/390, z/VM, VSE and Linux for S/390 and zSeries.

At the last SHARE, we set a goal to be able to build real-world 31-bit z/OS C programs by this SHARE. This goal has been accomplished. There is already enough of the infrastructure in place to build real z/OS UNIX System Services programs. The next goal is to improve LIB390 enough to perform I/O against MVS data sets, allowing non-USS programs to be built. At the same time, compatibility support is being added to LIB390 for MVS 3.8.

This is not a toy. GCC is a great mainframe compiler for both Linux and MVS!

How to Get Started?

GCC:
http://www.gnu.org/software/gcc/index.html

LIB390:
http://www.tachyonsoft.com/lib390.html

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

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

Or write to: David Bond at dbond@tachyonsoft.com