C: The “Dark Side” of System z?

Brandon Tweed
CA Technologies

March 2, 2011: 9:30 AM-10:30 AM
Session 9021

Target Audience

- Seasoned developers wishing to find a starting point with UNIX and C on z/OS
- Those wanting to learn more about how to use z/OS UNIX to build programs written in C
- A crash course in compiling, linking, Makefiles, and C under the z/OS UNIX environment
- Seeking an answer to the question “How do I make my college hire productive right away?”

Using a Star Wars Analogy

- Jedi – The Good Guys (Obi Wan)
- Sith – The Bad Guys (Darth Vader)
- Padawans – Young Jedi = Young Mainframers
- The “Light Side”
- The “Dark Side”
- Just Black and White?

The “Light Side”

- The classic approach toward mainframe software development
- Solving problems with ***patience*** and ***applied wisdom***
- Take the time to solve a problem and consider ALL possible solutions to find the best one
- **THINK**, do the required research, build the solution
- Don’t deploy the solution until all known scenarios are covered

The “Dark Side”

- Eager to take the quick and easy way out
- Run off and build something quickly, refactor it later (rewrite it)
- Get your arm cut off in the process
- Don’t waste time/money considering all scenarios
- “Just get it to work” under the common scenarios
- Hasty and prone to failure
- “Agile Development” when done improperly

Just Black and White?

- From observation z/OS UNIX, C, and LE are avoided by experienced Mainframers
- These are merely tools (much like light sabers)
- They can be used for good or evil
- How can we use these tools for good?

Consider "Young Mainframers"

- These are "Padawans"
- Scripting languages (PHP, Python, bash, sh, etc.)
- Windows/Microsoft Tools and Languages
- UNIX Tools/Knowledge (man, bash, vi, pico)
- C, C++, Java, or .NET languages
- Distributed-style SCMs such as Subversion, CVS
- Make, ant, Maven for building code
- Maybe x86 assembler (not likely)
- Google everything!
- "What's a mainframe?"

Knowledge of a “Jedi Knight”

- ISPF for daily work
- PDS allocation/creation for source files
- The record-oriented structure of z/OS data sets
- Allocation attributes of source data sets, object decks, and load libraries
- JCL (Card types, PARM, symbolic, syntax)
- Job Management, Accessing JES Output (SDSF)
- Familiarity with how IBM writes and organizes doc, message formats, how to look up messages, etc.
- Much, much, much more…

Overcoming the Learning Curve

- Padawans have a LOT to learn before becoming Jedi Knights
- How can we make them useful without waiting for them to catch up?
- Give them the “Dark Sided” stuff to get started, allow them to learn other skills over time

z/OS UNIX: Part of the Solution

- Ability to work with familiar UNIX commands (the POSIX-compliant ones)
- Support for compiling and linking C (or C++)
- Other languages too, even HLASM!
- Built-in Make support
- Has most things that a distributed developer would need without learning a lot of the details

Overview

- Editing and creating source
- Example C program
- Compiling and Linking from USS
- Make – Building the software
- C – A Crash Course

Editing and Creating Source

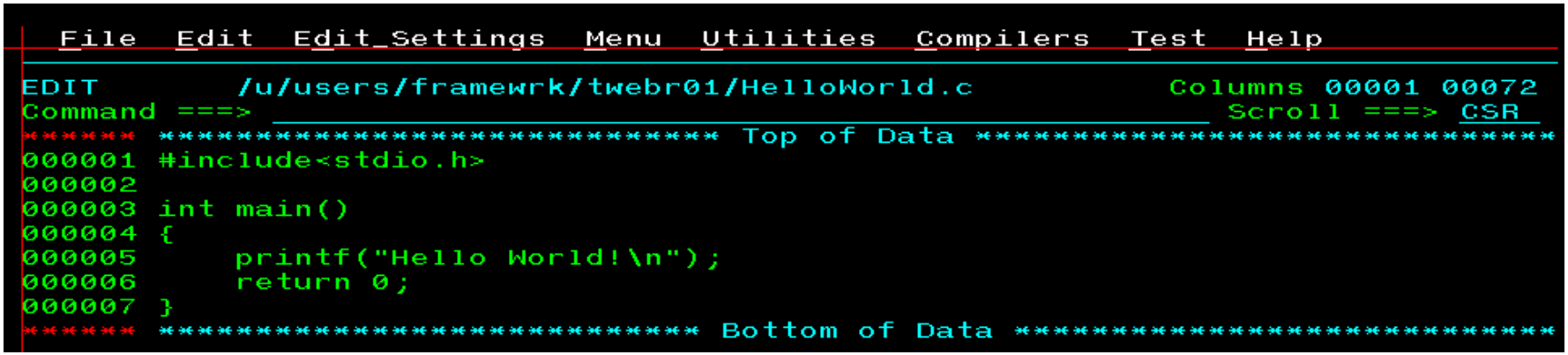
Choices:

1. Use OMVS or ISHELL combined with oedit and obrowse
2. Use a telnet client to connect to z/OS UNIX and vi as your editor
3. Use an editor that allows FTPing back and forth

Option 1: Editing With ISPF

- OMVS – A UNIX shell environment running underneath a 3270-style interface
- ISHELL – ISPF-style way of interacting with UNIX
- Use oedit or obrowse from the OMVS shell to view or edit source
- This is easier for ISPF developers, not as intuitive for new developers.

Option 1: Screen Shot

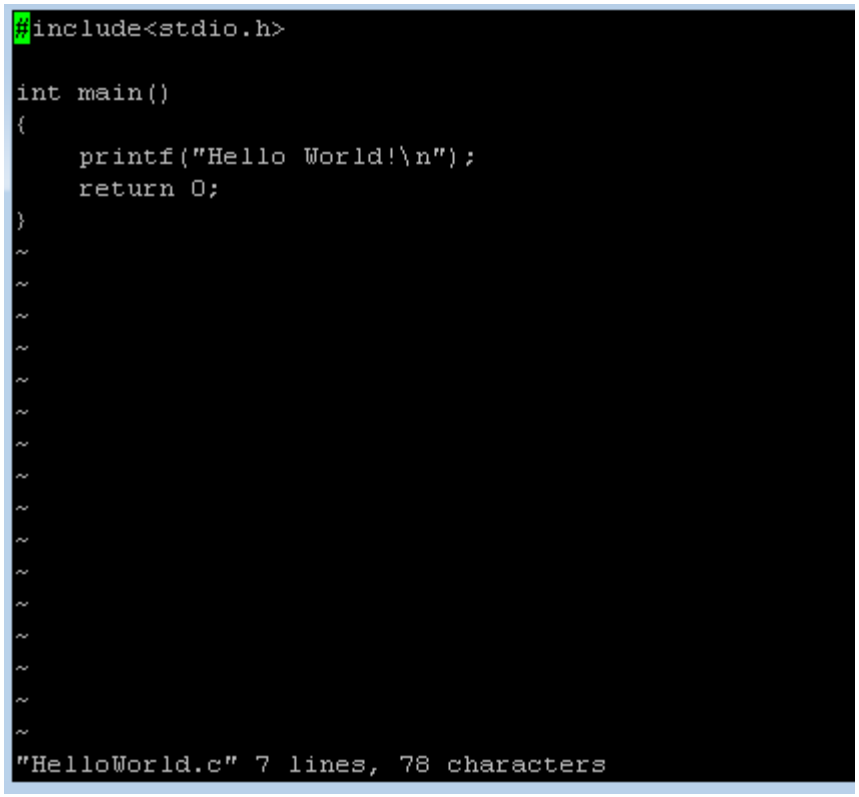


Option 2: Editing From A Telnet Client

- By default vi comes with z/OS UNIX
  - Command-driven
  - Non-intuitive interface to those that prefer GUIs
  - Powerful for those that love the interface

- This highlights the needs for more diversity in the editors available for z/OS UNIX

Option 2: Screen Shot



Option 3: Use An Editor with FTP

- A number of Windows-based text editors have built-in FTP support for uploading/downloading source
- Examples:
  - UltraEdit
  - Notepad++
  - Crimson Editor
  - WinSCP
- Requires FTP to be running/available
- Option 4: Spend money on ISPF replacements

Example C Program, HelloWorld.c

- Prints the message “Hello, World!”
- Message goes to console if running from z/OS UNIX
- Message goes to STDOUT DD if running under z/OS

```
#include<stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Compiling and Linking

Frame our discussion with how to build HelloWorld

1. Explain the mechanics of compiling
2. Explain the mechanics of linking
3. Explain how Makefiles help accelerate this process

Compiling from z/OS UNIX

- Translate source from human-readable format to object code format
- Output is an object code file or an “object deck”
- There is no “wall” between z/OS and USS

- Two possible paths:

1. Source in z/OS library yields object deck
2. Source in zFS or HFS produces a “.o” in zFS or HFS

Compiling Source to a PDS or PDS/E

- By default, compiler input matches the output
- Input from a PDS gives output to a PDS (or PDS/E)
- Compiler will assume that for libraries named *hlq.C* the output should go to *hlq.OBJ*
- The ".OBJ" library is created if one doesn't exist
- The input file needs to be encased in double quotes and preceded by slashes

Compiling Source to a PDS or PDS/E (Continued)

- Sequence numbers will cause problems for the compiler!
- Edit source with NUM OFF

- Example Source Member: TWEBR01.SOURCE.C(HELLO)
- Output Library (Implicit): TWEBR01.SOURCE.OBJ
- Example UNIX command:

c89 –c “//’TWEBR01.SOURCE.C(HELLO)’”

Compile to HFS or zFS

- Generally easier to manage
- Source is a “.c” file in the current directory
- Output is a “.o” file in the current directory

- Example:

c89 –c HelloWorld.c

Linking - Overview

- Input to the linker can come from
  1. ".o" File
  2. An object deck (in PDS or PDS/E)

- Output can go to:
  1. Executable file in zFS or HFS
  2. PDS or PDS/E

- This means there are a good number of possible combinations

Linking – Input From An Object File

- c89 needs the name of the input files
- It needs the name of the output files

Linking an object file to a PDS:

c89 -o "//'TWEBR01.SOURCE.LIB(HELLO)'" HelloWorld.o

Linking an object file to a UNIX executable:

c89 -o HelloWorld HelloWorld.o

Linking – Input From An Object Deck

- Object deck is stored in a PDS or PDS/E
- Want to link it to either a PDS or UNIX
- Linking an object deck to UNIX executable is uncommon

Object Deck to Load Module:

```
c89 -o "//'TWEBR01.SOURCE.LIB(HELLO)'" "//'TWEBR01.SOURCE.OBJ(HELLO)'"
```

Object Deck to UNIX executable:

```
c89 –o HelloWorld “//’TWEBR01.SOURCE.OBJ(HELLO)’”
```

Useful Compile Options

- Compile with “-g” to add debugging capability
- Get pseudo-assembler output with LIST option
- Use “-W” to provide additional linking or compiler options

Using the LIST Compiler Option

- Provides a “pseudo-assembly listing”
- The listing is basically a line-by-line translation of the C code into assembler code.
- Useful for dump reading when problems can’t be solved by looking at a traceback

Example Usage:

c89 –Wc,”LIST(HelloWorld.lst)” HelloWorld.c

More Information On Compiling and Linking

- z/OS XL C/C++ User's Guide
- See "Compiling and binding in the z/OS UNIX System Services environment"
- UNIX man command (the "man pages")
- man c89

“Make it so!”

- UNIX command “make”
- Typically used to build C, C++, or other source in a UNIX environment
- Students typically exposed to this in a CS degree
- Think of this as being analogous to a large JCL used to compile a program
- Only re-compiles or re-links if source has been modified (based on UNIX modification timestamp)

Anatomy of a Makefile

- A Makefile is a file that is provided as input to the “make” command
- By default “make” looks for a file named “Makefile” in the current directory if no input file name is provided to the command
- Makefiles are composed of **rules** and **macros**
- Rules define each thing that needs to be made and what that thing depends on
- Macros are like variables
- Comments start with a “#”

Example Makefile

```
# CC is a macro
CC=c89

# The following is the rule for target HelloWorld
HelloWorld: HelloWorld.o
        $(CC) -o HelloWorld HelloWorld.o

# The following is the rule for target HelloWorld.o
HelloWorld.o: HelloWorld.c
        $(CC) -c HelloWorld.c

# The following is the rule for target HELLO
HELLO: HelloWorld.o
        $(CC) -o "//'TWEBR01.SOURCE.LIB(HELLO)'" HelloWorld.o
```

Macros

- General format:

  MACRO_NAME=value
- In the Makefile use $(MACRO_NAME) where you want the value substituted
- You probably noticed the CC=c89
- The symbol CC is a macro
- $(CC) will be replaced with the value c89
- UNIX environment variables may also be referenced like macros

Breakdown of a Rule

- A rule consists of a ***target***, ***prerequisite***, and a ***recipe***
- The **target** represents the thing that will be built by this rule
- The **prerequisite** is the list of things that need to be built (or already exist) before the target can be built
- The **recipe** is one or more lines of UNIX commands that are used to create the target

Example of a Rule

```
HelloWorld.o: HelloWorld.c
     $(CC) -c HelloWorld.c
```

- Target is HelloWorld.o
- Prerequisite is HelloWorld.c
- The recipe is the command c89 (remember our macro?)

More Details About Rules

- The recipe can be more than one line
- Each line of the recipe MUST begin with a tab character, X'05'
- If using oedit,
  1. can use hex editing to add the tabs
  2. Use another character, do a CHANGE ALL
- The list of prerequisites can consist of many items

The Key: Rules Depend on Each Other

- Based upon the rules, make has a lot of work to do
- To build a target for a specific rule:

```
for each prerequisite
    if the prerequisite does not exist then
        find the rule where prerequisite is a target
        process that rule (Recurse)
        if rule was not able to build target, quit
    endif
end for
issue the recipe commands to build the target
```

Another Look!

```
# CC is a macro
CC=c89

# The following is the rule for target HelloWorld
HelloWorld: HelloWorld.o
        $(CC) -o HelloWorld HelloWorld.o

# The following is the rule for target HelloWorld.o
HelloWorld.o: HelloWorld.c
        $(CC) -c HelloWorld.c

# The following is the rule for target HELLO
HELLO: HelloWorld.o
        $(CC) -o "//'TWEBR01.SOURCE.LIB(HELLO)'" HelloWorld.o
```

Tips for Building Makefiles

- Remember to keep your TAB characters at the beginning of your recipe lines
- Make sure your editor formats the Makefile with LF character at the end of the line
- Lines ending in CR/LF probably OK
- Lines ending in LF/CR will cause problems, particularly in recipes

So Far…

- Compiling
- Linking
- Makefiles
- Everything needed to build the source

What about the CODE!?

C: The Language

- Overview
- Data Types
- Basic Program Structure
- Source Structure
- The pre-processor
- Troubleshooting

C – An Overview

- Compiler-optimized, native code = fast performance (especially compared to Java)
- Includes an extensive run-time library
- Can be used to invoke HLASM code (a whole session in itself)
- Can be invoked by assembler using CEEPIPI (there's a session about CEEPIPI!)
- benefits of low-level code without the minutiae
- Simplicity!
- CS students typically know this coming out of school

C – Data Types – Primitive Types

- C is a “typed” language.
- Data falls into basic categories
- Sizes depend on machine architecture or AMODE
- int (long, short, unsigned) – Integer data, can be signed or unsigned. Generally the size of the register for the AMODE.
- char (signed or unsigned) – A byte of data.
- float, double – Floating point or double-precision floating point.

C – Data Types – Pointers

- Pointers – referential data type
  - Think “pointer = Address”. Yes, it’s that simple.
  - Generally a pointer is the same size as the register for the AMODE. This makes sense if you think about it.

Examples:

int i = 5; // set the integer i to the value 5

int * iptr = &i; // iptr is a pointer set to the address of i

C - Data Types – Structures

- Structures are conceptually the same as a control block
- They have fields (individual variables)
- Variables are laid down in memory in the order they are declared
- The following structure named “my_struct” stores three integers, named “a”, “b”, and “c”.

```
struct my_struct {
        int a, b, c;
}
```

C – Data Types - Arrays

- To define an identical set of items laid down contiguously in memory use an array:

  char buffer[256];

- This defines 256 characters laid down in memory. Each character can be referenced by providing an index.
- buffer[0] is the first byte. buffer[255] is the last.
- Not restricted to built-in data types. You can create arrays of statically-allocated structures.
- You can also dynamically allocate and construct your own arrays.

C – Data Types - Strings

- A “string” in C is a sequence of characters in memory terminated by a NULL byte (value X’0’).
- Different from z/OS common practice (uses lengths)
- Declare an array of characters
- Always allocate an extra byte for the NULL

Basic Program Structure

- A “vanilla” C program has a primary entry point called “main”
- main returns an integer value (the return code of the program)
- main contains all of the C statements between curly braces
- the “parameter” list passed to main is found between the parentheses

Example

```
int main()
{
  int i;
  i = 0;
  return i;
}
```

- Declares an integer variable, i
- Sets that variable to 0
- returns to the caller
- End result is RC=0

What About Other Routines?

- The C terminology for a “routine” is “function”
- Functions have a similar format to main
- They can return a value, accept a parameter list, and perform processing
- main usually invokes other routines.

Program Structure - Another Example

```
int increment(int i);  /* This declares the name of the function to the compiler */

int main()
{
      int i = 0;
      i = increment(i); /* Main calls the function */
      return i;
}

int increment(int i) {
    return i+1; /* The function increments the value and returns it */
}
```

Structure

- The example will give an RC=1
- A function can invoke itself for recursion
- External code such as HLASM or other types of modules can be invoked using syntax outlined by LE manuals
- Some programs don’t have a main, typically DLLs.
- DLLs package re-entrant functions that are reusable

Source Structure

- C is conventionally broken into two parts:
  1) source that defines the actual function implementation (“.c” files)
  2) source that defines variables, functions, and symbols that can be “re-used” (“.h” or “header” files)
- The “#include” directive can be used to pull a header file “into” the current source file. Similar to the COPY verb.
- C compiler translates the code using a number of passes.
- The “preprocessor” is the pass that handles includes and other directives

Run-time Library

- Dynamic memory allocation
- Codepage translation
- I/O – to UNIX files or data sets
- Security routines
- Sockets API
- Threading
- string manipulation

Debugging/Troubleshooting

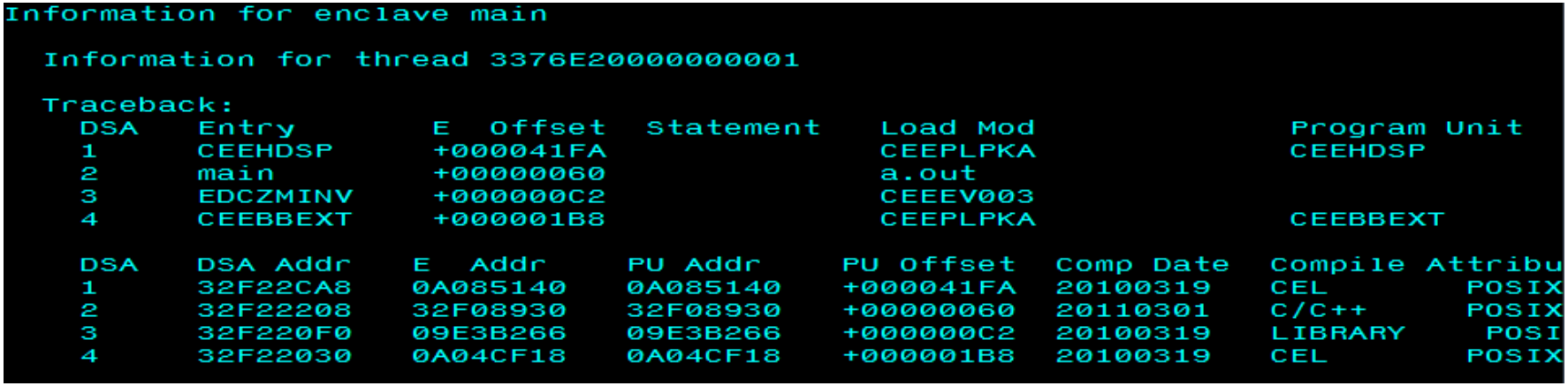
Common Exceptions:

- S0C1 – Operation Exception
  - Happens when you try to execute code in an area with invalid instructions
  - Usually means a pointer to a function was dereferenced that had an invalid value or pointed somewhere it shouldn't
- S0C4 – Protection Exception
  - Tried to reference storage that was protected
  - 90% of the time, you dereferenced a NULL or uninitialized pointer (you referred to address 0!)

Traceback

- Found inside the dump produced by LE typically when an “unhandled condition” happens
- If running in batch, include CEEDUMP DD in the job
- If running in UNIX, output will go to a CEEDUMP.* file in current directory
- Gives you an idea of what the function call stack was and usually the name of the function where the exception happened

Traceback Example



```
Information for enclave main

  Information for thread 3376E20000000001

  Traceback:
    DSA    Entry        E  Offset  Statement   Load Mod              Program Unit
    1      CEEHDSP      +000041FA               CEEPLPKA              CEEHDSP
    2      main         +00000060               a.out
    3      EDCZMINV     +000000C2               CEEEV003
    4      CEEBBEXT     +000001B8               CEEPLPKA              CEEBBEXT

    DSA    DSA Addr    E  Addr     PU Addr     PU Offset  Comp Date  Compile Attribu
    1      32F22CA8    0A085140    0A085140    +000041FA  20100319   CEL         POSIX
    2      32F22208    32F08930    32F08930    +00000060  20110301   C/C++       POSIX
    3      32F220F0    09E3B266    09E3B266    +000000C2  20100319   LIBRARY      POSI
    4      32F22030    0A04CF18    0A04CF18    +000001B8  20100319   CEL         POSIX
```

In Summary

- Compiling and Linking – Intuitive when done from UNIX for a Padawan
- make – Defer the need to teach details of JCL, automate construction
- C – A powerful starting language with simple syntax, an extensive run-time library, and low-level capabilities

Closing Thoughts

- z/OS UNIX and C are tools, capable of being used for good or evil
- System z, like any platform has both “accidental complexity” and “essential complexity” (Using Fred Brooks terminology)
- Starting learners with UNIX helps address the “accidental complexity” issue
- Use UNIX and C as a starting point for teaching the ways of the Jedi and the “Light Side” of the Force
- C and UNIX are only the “Dark Side” when used for evil

“May the Schwartz Be With You!”

- Questions? Answers?