Abstract:
This paper discusses first time user experiences on the IBM mainframe. The specific areas that will be covered are:

- Linking Multi-language programs

Usage of the INDep option:
The INDep compiler option has two uses: first is to allow C code to be called directly from other high-level languages. Second, is to allow C code to execute in the absence of the C runtime library.

- Compile all C code that may be called from other languages with INDep and Linkedit normally, whether the function is named main() or not.

- C code compiled with INDep can be called from FORTRAN, COBOL, PLI, or any other language that uses standard IBM linkage conventions.

- Know that many high level languages require their own execution framework (which is a collection of data and routines supporting the execution code).

- When control passes to C, the C framework must be active, therefore, the appropriate framework must be active whenever control passes across a language boundary.

### Linking Multilanguage Programs

Usage of the INDep option:
The INDep compiler option provides two uses:

- First, it allows C code to be called directly from other high-level languages.
- Second, it allows C code to execute in the absence of the C runtime library.

- Compile all C code that may be called from other languages with INDep and Linkedit normally, whether the function is named main() or not.

- C code compiled with INDep can be called from FORTRAN, COBOL, PLI, or any other language that uses standard IBM linkage conventions.

- Know that many high-level languages require their own execution framework (which is a collection of data and routines supporting the execution code).

- When control passes to C, the C framework must be active, therefore, the appropriate framework must be active whenever control passes across a language boundary.
The SAS/C Source Level Debugger is a useful tool to aid in development and testing programs compiled using the SAS/C Compiler.

Capabilities:
- Running interactively under TSO and in batch.
- Using Debugger commands and command arguments, you can get information about what is occurring at various points in an executing program (trace program flow).
- Perform a variety of actions to access and modify objects, including structures, unions and arrays. Assigning values to objects as well as displaying the content.
- Conditional breakpoints and actions.

Getting Started Running Under TSO:
- Compile all parts of your program that you want to debug with the -Debug option.
- Allocate a data set for the debugger symbol table file. The data set will contain a member for each compilation. The member contains debugging information for that compilation which includes the name of the source. The data set should also be allocated with a retnum, NRENUM=4000 and associated with a data set with the SYSDLIB DName at compile time and the DBGLIB DName at run time.

The SECTION-NAME is a section name for each source file as specified with the SName compiler option. The debugger uses the section to locate the member in DBGLIB that contains debugging information for the section. If you compile with the SName option, the member name in the symbol table file is the section name you specify. Otherwise, the member name is the default section name.

Invoke the debugger:

```
PROC D DIAG MEMBER(T)
/*
  / TO EXECUTE PROC IN NATIVE TSO
  */
*/
/* CLIB DIAG MEMBER ASSEMBLATION PARK
  */
/* CLIB DIAG MEMBER ASSEMBLATION PARK
  */
CONTROL NORM NOCLIST NOCLIST MAIN
ERUBF RETURN CLIST ERRORS */
IF ADIJAG ED TSO THEN DO
  CLSCON /* DIAGNOSTIC MODE */
  WRITE RETROS ASYSLONG ENTERED */
  CONTROL NOS LIST CON1LIST
END
FREE FI (CPILIB)
ALLOC FI (CPILIB) DA('USER.PDS.LINKLIB') SHR
FREE FI (DBGLIB)
ALLOC FI (DBGLIB) DA('USER.C.DEBUG(MEMBER or MAIN)') SHR
FREE FI (DATA1N)
ALLOC FI (DATA1N) DA('USER.INPUT.DATA') SHR
C MEMBER =0
```

Under TSO, you can use the TSO CALL command and pass the parameter "d."

call t527x.pds.lnklib(cable) "d"

Alternatively, you can call your program under TSO as a command processor, such as:

c cable =d

A detailed description of debugger commands and usage information are outlined in the SAS/C Source Level Debugger User's Guide.

The -Debug option will suppress code optimizations and increases the size of the object code for your program. Once the debug test determination is complete, recompile without the -Debug option.
3. Run time Library Option to Optimize Programs

With SAS/C Compiler 3.01, our development staff have been able to improve the performance of programs that are heavily CPU bound by means of applying run time parameter -optimize or -minimal.

-optimize specifies that you want an optimized form of program linkage. When this option is specified, function call overhead is decreased. However, there are some restrictions that must be followed listed below.

-minimal specifies that a minimal form of program linkage is desired. Use of minimal linkage is recommended only for thoroughly tested and reliable programs for which performance is critical. When minimal linkage is requested, a single area of memory is allocated when the program starts up. Overflow of this area is not checked. If over flow occurs, random ASENDS or over lays of the program or other data are expected. Note: do not use this form of linkage for programs using recursive algorithms unless you know the upper bound to the amount of recursion required. For this reason, the library call routine should not be used with -minimal. Use of this option minimizes the overhead of function calls, generally producing significant savings. However, there are restrictions described for optimized linkage such as:

- Only C and assembler language subroutines are permitted.
- Both are mutually exclusive.
- Functions compiled with the INDep option cannot be used.

Optimized linkage is recommended for production programs that meet these restrictions.

Three datasets containing (50,16,120) members each.

<table>
<thead>
<tr>
<th>Enhance</th>
<th>Total CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>memopt</td>
<td>102</td>
</tr>
<tr>
<td>opt</td>
<td>173</td>
</tr>
<tr>
<td>ssp</td>
<td>162</td>
</tr>
<tr>
<td>New 'B'</td>
<td></td>
</tr>
<tr>
<td>filter</td>
<td>140</td>
</tr>
</tbody>
</table>

Benchmarking of Assembler, C, PL/I

A project was initiated to compare assembler, C, and PLI on the IBM 4381 using a production application for CPU efficiency. A subroutine which has many applications in offline projects was identified. Programs were written in Assembler-H version 2 release 1, SAS/CO Compiler 3.01H, PLI Optimizing compiler version 1 release 5.1. The programs had a driver program and three subroutines. Each program was optimized for efficiency after being run with the software package STORF, tested by dumping memory, analyzing maps, and compared for differences in programming logic and style. The differences were eliminated as much as possible.

Each program was run in a stand alone environment using 127984 records that were mapped and moved.

<table>
<thead>
<tr>
<th>Percentage of CPU time spent in each module</th>
</tr>
</thead>
<tbody>
<tr>
<td>subroutine</td>
</tr>
<tr>
<td>routine ver 2</td>
</tr>
<tr>
<td>rel 1</td>
</tr>
<tr>
<td>DRIVER</td>
</tr>
<tr>
<td>MAPSUBF</td>
</tr>
<tr>
<td>MAPSUSF</td>
</tr>
<tr>
<td>MOVEFLO</td>
</tr>
<tr>
<td>Tot CPU</td>
</tr>
<tr>
<td>Exec time</td>
</tr>
</tbody>
</table>

In terms of CPU efficiency, Assembler clearly out performed the two compilers. This only stands to reason since all compilers generate machine code and there is a one to one correspondence between each assembler line of code and machine code. Of course, it is possible for an inexperienced assembler programmer to generate less efficient code than from an experienced programmer using PLI or C.

C generated relatively efficient code versus PLI when looking for subfield definitions in subroutine MAPSUBF using 26.91% CPU time compared to PLI using 73.92% CPU time. C's efficiency in MAPSUBF was able to overcome the relatively less efficient code generated in the DRIVER program versus PLI. C spent less amount of CPU time in subroutine MAPSUSF and MOVEFLO than both ASM and PLI. The technique of register variables did not improve CPU usage in this benchmark.
In discussion with technical support at SAS Institute, two reasons for the high percentage of time in the DRIVER routine was explained.

1. I/O is UNIX-based and is relatively inefficient compared to the other languages. C I/O must go through several layers before physical I/O is done while PL/I I/O is close to physical I/O.

2. C has quite a bit of overhead involved in calling subroutines.

If one is only concerned with CPU efficiency, Assembler is the language of choice. However, other factors are clearly involved, including but not limited to, availability of programmers in specified languages, ease of maintenance, stability of the compiler, portability, and ability to use structured techniques.

4. Porting Programs and ANSI C Standards

As C language has grown in recent years, one of the attractive features that C language has to offer is its ability to be ported from implementation to implementation. Even though each implementation may differ to some degree, it may be a good practice to obtain as much documentation on the implementation and libraries before starting to ensure portability.

By adhering to the ANSI C Standards (American National Standards Committee X3J11, the ANSI Technical Committee which has been appointed to develop C Language Standards) provided for C language, users may enhance the reliability, maintainability, and execution efficiency when porting C language to various computer systems.

SAS/C/C features from the ANSI Draft will be extensions to the language defined by Kernighan and Ritchie (1978), or that differ significantly from Kernighan and Ritchie, are described in the SAS/C Compiler and Library User's Guide 3.01. Areas where the SAS/C/C compiler currently differs from the ANSI Draft are also discussed. With SAS Technical Report C-106 (Changes and Enhancements to the SAS/C Compiler 4.00), there is more discussion on how to take advantage of many of the improvements now available over the original Kernighan and Ritchie description of the C language. It also discusses terminology and organization of the Draft Standards.

Experience:

Memory space - will the program ported require more memory than the capacity of the target system?

Evaluate all system dependencies and memory requirements before porting large programs.

Libraries - in general, most library functions are quite compatible, albeit there may be some differences that will have to be resolved if complete portability is to be achieved. One solution is to define your own function using the preprocessor.

The process of porting can be made simpler if you have access to some standard programming tools such as grep (pattern checker or string of char), lint (type and syntax checker), and a make file (to keep track of interdependencies). To accomplish a grep (pattern checker), you can use USPF 3.14 SEARCH-FOR UTILITY, specifying a data string. For version control, you can use USPF 3.12 SUPERB UTILITY, when comparing two libraries or members for any changes made and indicating the differences.

Signed or Unsigned Char - depending upon what implementation, a char can be either signed or unsigned. When evaluating this expression:

#define EOF -1
char c; /* signed or unsigned */
while ((c = getchar()) != EOF)

The function getchar returns an int type.

If char is signed the range may be from 0 - 255 causing the while loop to never end.

Identifiers - external identifiers, whether they are names of functions or variables can create problems when porting from implementation to implementation. There may or may not be a limitation on the maximum number of characters allowed to be significant.

The first function name in the source program should be the member name when linking several programs together. If D*names are referenced, indicate them as eight characters or less. Truncation and warning messages are generated if D*name exceeds eight characters.

Interactive programs - when porting interactive user programs, each screen display may differ to some degree on a given computer system. Considerable care should be taken to ensure screen displays remain consistent as well as functional.

Beware of Warning messages when porting - resolve all warning from one implementation and vice versa. A warning on one implementation may be an error on another.
function Prototypes - to ensure complete correctness of argument types and the number of arguments passed during a function call, a common practice is to provide function prototypes within a library header file.

PROTOTYPE EXAMPLE

```c
#include <stdio.h>

int fune(long x, short y);

void main()
{
    long a = 32768;
    short b = 10;
    int answer;
    answer = fune(a,b);
    answer = fune(b,a);
    int fune(x,y)
    long x; short y;
    { printf("x = %ld, y = %d-n", x, y);
    return(0);
    }
}
```

Results using Microsoft C Optimizing Compiler 5.1
x = 32768, y = 10
x = 10, y = -32768

Results using SAS/C Compiler 4.00
x = 32768, y = 10
x = 10, y = -32768

```c
#include <stdio.h>

void main()
{
    long a = 32768;
    short b = 10;
    int answer;
    answer = fune(a,b);
    answer = fune(b,a);
    int fune(x,y)
    long x; short y;
    { printf("x = %ld, y = %d-n", x, y);
    return(0);
    }
}
```

Results using Microsoft C Optimizing Compiler 5.1
x = 32768, y = 10
x = -2147483638, y = 0

Results using SAS/C Compiler
x = 32768, y = 10
x = 10, y = -32768

Conclusion

In short, I have briefly discussed first-time user experiences in the following areas: Linking multilanguage programs, where the important point is to beware of the execution framework that must be created prior to language calls or returning from another. SAS/C Compiler 4.00 Interlanguage Communication feature User's Guide provides excellent documentation as well as examples on this topic. The SAS/C Debugger can be a very useful tool to aid in development and testing of programs, providing numerous debugging command arguments and commands. To improve the performance of heavily CPU bound programs, try using the run time options maximize or minimal options to improve CPU utilization. When porting C language code, it would be a good practice to obtain as much documentation on the implementation as possible, and look for system dependencies. Because of the slight differences from implementation to implementation, porting is not a simple task. The adopting of the ANSI C standards is a way to make these differences smaller and less significant.

Acknowledgements
The author would like to thank Mr. Dan Schumacher and Mr. John Harry for their assistance at OCLC Online Computer Library Center, Inc.

SAS/C Compiler is a trademark of SAS Institute Inc., Cary, NC, USA

IBM is a registered trademark of International Business Machines Corporation, Armonk, NY, USA

STROBE is a registered trademark of Computer System Architects Incorporated.

Microsoft is a registered trademark of Microsoft Corporation

UNIX is a registered trademark of AT&T Bell Laboratories.