Systems Programming Applications In C Using the SAS/C® Compiler

Ingrid Ammondson, SAS Institute Inc., Cary, NC

ABSTRACT

Traditionally, systems programming for the IBM® 370 has been performed using assembler. This is because systems applications require a language that provides efficiency, control over the environment, and access to system data and services. On UNIX® type operating systems, the C language is the systems programming language because C has features that make it suited to systems programming applications. With the development of C compilers for the 370 environment, it is desirable to write 370 systems programs in C in order to gain the programming and maintenance benefits inherent in a high-level language. This paper addresses using C for 370 systems programming applications, the problems involved, and some possible solutions to those problems.

INTRODUCTION

The C language is a relatively new programming language, with the fastest-growing user base of any language. It was developed in 1972 at Bell Laboratories by Dennis Ritchie and has been traditionally used for systems programming applications in the UNIX environment. The language is block-structured, facilitating the development of well-formed, structured programs. C contains many useful data types and a wide set of operators, which, along with its rich library, make it extremely powerful. Because C code is compact and supports basic hardware data types and operations, it is an efficient language compared to other high-level languages such as PL/I. C is standard in most university computer science curricula, and there are many C programmers available. A growing body of software is written in C, much of it portable. Good C compilers exist on most machine architectures. The draft ANSI X3J11 C Standard is close to becoming final, providing a standard definition for C.

In the past three years, with the availability of good C compilers for the IBM system 370, the use of C on the 370 has increased significantly. Today, there are a large number of sophisticated applications written in C available for the 370 environment, including system tools and utilities, performance systems, expert systems, and engineering, financial, and information retrieval systems.

One of the newest uses for C on the 370 is for writing systems programming applications. Writing systems applications in C is desirable in order to gain the programming and maintenance benefits inherent in a high-level language. This paper describes the nature of 370 systems programming applications and the language requirements for such applications. The suitability of C for these projects is examined, and guidelines for using C are discussed. The support for systems applications provided by the SAS/C® compiler also is described, and example applications presented.

CHARACTERISTICS OF SYSTEMS APPLICATIONS

Categories of Systems Applications

Traditionally, systems programming for the IBM 370 has been performed using assembler. Because there is little literature or training available on writing systems applications, it is useful to attempt to characterize these applications. The following categories of systems applications can be considered:

- exits from operating system components, such as VTAM or JES2, or from other software, such as RACF or ISPF
- extensions to the operating system, such as user supervisor calls (SVCs) or new CMS commands
- low-level applications, such as shared file servers or performance measurement tools, which require direct access to the features of the hardware and operating system
- programs that must execute partially or entirely asynchronously, such as real-time or event-driven applications.

Popularity of Software Exits

User-written exits from all types of software are becoming popular, partly due to the trend towards Object Code Only (OCO) software distribution. Most users accept OCO distribution as long as user modifications to the software are still possible. Software exits can be an elegant solution for providing system flexibility and allowing user customization. Many vendors are incorporating exits into their products.

Another reason for the increasing popularity of exits is that this allows software vendors to write packages that are portable, with system-specific capabilities provided through exits. Exits can be found in virtually all types of software, including production control systems, automated computer center managers, performance monitors, software management packages, and operating systems and utilities.

Diversity of Systems Applications

Similar types of applications may have diverse characteristics in such areas as complexity, frequency of invocation, and environmental constraints. To illustrate this diversity, consider the following types of exits:

- an exit that can be invoked at any point, including at system initialization time, and that cannot depend upon the full availability of operating system services, such as a DASD management exit
- a JES2 exit for scanning a job card that is executed once per job and contains only 30 assembler statements
- a RACF exit that is executed each time any data set is opened and whose performance is therefore critical.

Constraints upon Systems Applications

Despite their diversity, systems applications must usually execute within constraints not imposed upon other types of applications, due to their close association with the hardware and operating system.

Efficiency

Systems applications, particularly extensions to the operating system, must be efficient and compact. Often these routines consist of short code sequences that are stored in shared memory and executed many times, such as a CMS end-of-command nucleus extension or a RACF exit to provide fast path access to data sets. Some exits, such as SMF exits, involve a large amount of processing and recording of data.

Environments

Exits may be required to execute in the environment of the product that invoked them, such as JES2 or CICS. This can place restrictions and requirements upon the exits in such areas as linkage conventions, usage of service routines, and communication protocols.

Linkage Conventions

Applications that must conform to the environment of the calling product may be called with a wide variety of nonstandard and inconvenient linkage conventions. Such applications may include JES2 or CICS exits, CMS nucleus extensions, interrupt handlers (such as STIMER exits), or extensions to the operating system (such as MVS SVCs).

Control Block Access

System control blocks provide a means for systems applications to communicate with hardware, components of the operating system, and other applications. These control blocks are frequently defined through assembler DSECTs in libraries provided as part of a product. The information contained in control blocks is normally bit flags, counters, other status-type information, and the addresses of other control blocks. Control blocks are often linked together in queue or tree structures. Efficient construction and navigation of these structures and the manipulation of the data within are requirements for many systems applications.

Parameter Blocks and Return Codes

In addition to system control blocks, information is passed by systems applications via parameter blocks and return codes. There are often stringent and environment-specific rules for this means of passing information.

System Macros and Service Routines

Virtually all systems applications perform standard actions, such as entry to and exit from the application, and request system services, such as memory allocation. Macros and SVCs are used to perform these actions. The macros may be supplied as part of an operating system or software package, or they may be part of a user-written service library. Exits that run in some environments may be required to use only the macros and services provided by that environment, such as a CICS task-related user exit that must issue CICS GETMAINs rather than OS GETMAINs.

Execution Modes

A systems application may execute as a normal problem-program or may be required to execute in a privileged environment. There may be dependencies on addressing mode, privilege level (such as protect key), and other execution attributes (such as running in a disabled state). Because they may execute in a privileged mode and can therefore damage the operating system if they fail, systems applications must be coded with greater caution than other applications.

Because no high-level language has been available that can uniformly meet the requirements of applications with the above characteristics, documentation for various product exits and systems software often states that assembler must be used. To illustrate that C is a viable language for such applications, it is useful to examine the requirements for a 370 systems programming language.

REQUIREMENTS FOR A SYSTEMS APPLICATIONS LANGUAGE

A high-level language implementation is composed of both a language translator and a library of run-time support routines. The nature of systems applications places a unique set of constraints and requirements on the language and its library.

Language Requirements

The language requirements for a 370 systems applications language and their relationship to C are examined in the following paragraphs.

Address Access and Operations

Perhaps the most important language requirement for a systems application language is simple and fast access by address to control blocks, data, and code and the ability to perform operations on such addresses. C provides the pointer data type, the address operator, and pointer arithmetic. The implementation is straightforward, easy to understand, and closely related to actual hardware operations.

Locating and manipulating other bodies of code are systems application requirements. C function pointers are extremely powerful for this purpose, enabling the location of a function to be manipulated as a variable while the actual address of the function need not be known. For example, an array of function pointers can be declared and initialized and passed to other load modules to enable the functions to be called by and shared between several programs.

Bitfield Definition and Manipulation

Systems applications frequently require definition and manipulation of bitfields, for example when interfacing to hardware such as I/O devices. The C language provides operators to access and manipulate bits contained within a word. Additionally, partial words may be treated as bits through the C bitfield data type.

Table Searches and Parameter Parsing

Functions such as table searches and parameter parsing are frequently required in systems applications. C provides variable-

length string-handling support and includes library functions that are useful tools for building routines to perform such operations as command parsing and message building.

Macro Language

There are many programming functions that are routinely performed in systems applications. A macro language is a powerful tool for creating reusable code and avoiding the overhead of a subroutine call. For example, certain code sequences, such as those required to issue an ENQ or a GETMAIN, are often repeated with minor variations. A macro language can allow the basic pattern of these code sequences to be defined and saved in some manner and then expanded and executed.

C contains a macro preprocessor that supports symbolic name replacement, definition of commonly used code sequences, inclusion of text from separate files, and conditional compilation. Conditional compilation allows code sequences to be varied at compile time, depending upon a condition known at compile time, such as the operating system under which the code will execute.

Nonpaternalistic Language Characteristics

A general language characteristic that is a requirement for a systems application language is that the language be nonpaternalistic — that is, that the language not interfere with or prevent potentially unsafe coding practices. For instance, while using zero as a pointer value is ordinarily a mistake, a CMS systems application may need to reference the contents of low memory in this way. C allows such operations.

A unique feature of C is the approximate equivalence of pointers and arrays. For example, a pointer variable can be subscripted to access items after the first one pointed to. Since arrays of unknown size are not supported, this enables pointers to be used as indexes in situations where a variable number of items might occur, such as in processing entries in the MVS TIOT control block.

In general, C conforms to the gun-bullet-foot style of programming — it's your gun, your bullet, and your foot. (Of course, when programming a systems application, it pays to be careful with such constructs. A single bullet, carelessly fired, can shoot the foot of everyone using the system.)

C Language Extensions

In addition to the standard language features as defined by the ANSI X3J11 C Standard, there are a number of features a compiler vendor can provide to enhance the utility of C for systems applications.

Much systems software is required to be reentrant, since it may be installed in shared memory and is normally executed by many users. While reentrancy is not an actual part of the C language definition, the SAS/C compiler provides a compilation option to generate completely reentrant code with no limitations on programmer coding style. This includes no limitations upon the initialization and modification of `static` and `extern` data.

The ability to map system control blocks using C structures is very important and sometimes requires the elimination of padding in C structures. The SAS/C compiler provides the BYtealign compiler option to align data elements within structures on byte boundaries. The alignment of individual structures can be controlled through the `__alignmem` and `_noalignmem` keywords. Additionally, an implementation of tightly packed partial-word bitfields is helpful for control block mapping. The SAS/C compiler provides a compiler option to treat bitfields as `char` rather than `int` to assure maximum packing.

The generated code for a systems application must be efficient and compact. C is inherently a compact language. Additionally, each new release of the SAS/C compiler results in faster generated code. Release 4.00 includes a global optimizer phase to optimize the flow of control and data.

One important systems programming need that extends beyond standard C is the ability to control register allocation within the generated code. For example, it may be desirable to choose a particular register as a base register or to specify that a specific register not be used in the generated code. To meet all requirements for systems programming fully, a C compiler would need to generate assembler code as output, thereby providing control over registers, the ability to issue system macros, and the ability to conform to any unusual environmental requirements.

A degree of control can be achieved by providing the ability to write inline machine code from within C source. The SAS/C compiler provides this feature. This allows operating system SVCs and 370 assembler instructions to be issued directly from C, avoiding the overhead of function calls. Additionally, C macros can be defined to issue the sequence of machine instructions necessary to duplicate many assembler macros.

Library Issues

One of the advantages of using a high-level language is the presence of a library of support routines. The following paragraphs discuss requirements and desirable features for a library used for systems programming applications and the problems involved with using a standard C library for these applications.

Library Requirements

A library used for systems applications must be fast, compact, and nonintrusive. The functions provided should include support for basic programming functions, such as I/O, string-handling, and memory management. Support for specialized systems applications and requirements, such as interrupt handling, is a desirable feature. Library front ends should be available to make it easy to issue common SVCs and system calls.

Source code should be provided for those routines that interact directly with the operating system to allow tailoring to meet special requirements. The ability to use services other than the standard ones provided by the operating system, such as a CICS GETMAIN rather than an MVS GETMAIN, should be provided. Library functions should not require dynamic loading, as this function may not be available in all environments. Additionally, library functions not called should not be included in the load module in order to minimize load module size. It is important to be able to execute C code with minimal library support for applications where C can be used as a high-level code generator.

The environment required by the library must be small, flexible, and modifiable, and so source code should be available for the environment support routines. For example, C supports automatic variables and recursive function calls and therefore requires a program stack of some sort. But no single method of stack allocation and management meets the requirements of all systems applications on an architecture with no hardware stack, such as the 370. Therefore, it is important that the source for prologue and epilogue routines, which manage the stack, be available and that the support provided by the compiler vendor be flexible enough to accommodate different implementations.

The library should not impose arbitrary restrictions upon the programmer, as systems applications can require operations whose meaning or utility are questionable in a traditional application, such as treating the code of a subroutine as data. Not all error-checking is inappropriate, but the library should avoid rejection of any construct that might be (or become) meaningful. For instance, it is reasonable to require that a GETMAIN macro specify a non-negative amount of storage. But it is not reasonable to restrict the subpool number to a fixed set, particularly since new operating system releases could introduce new valid subpools.

Library Problems

The use of a standard high-level language library can cause problems for systems applications. For example, consider an application that requires repeated allocations of memory. It is certainly preferable to call a library routine that already implements a storage management algorithm rather than to issue repeated calls to DMSFREE or GETMAIN, which can fragment available storage and cost significant overhead. While calling a library routine is desirable, a degree of control is required over the library for systems applications. For example, while the above application may require that storage be allocated from a particular subpool, most standard library storage functions do not provide subpool selection as an option. Or perhaps the storage management algorithm used by the library does not suit the needs of the application. A normal high-level language library is often too inflexible for use in systems programming applications.

For C, the library problem is compounded by the close coupling of the ANSI definition of the library to the UNIX operating system. Many concepts from UNIX have no efficient counterpart, or even meaning, under the standard 370 operating systems. For example, C I/O functions assume that files are divided into records by newline characters encoded in the file. This assumption is technically incorrect for 370 file formats. A single record of a 370 object deck may contain any number of newline characters as data. It is not possible to create an efficient library implementation, such as is required for systems applications, that conforms to both architectures.

Furthermore, the ANSI library definition was created for general applications programming and includes requirements that add undesirable overhead for systems applications. For instance, ANSI requires that three standard files (`stdin`, `stdout`, and `stderr`) be opened by the library rather than by the application. For a systems application, this adds expensive and frequently needless overhead.

SAS/C Compiler Systems Programming Environment

The SAS/C compiler's Systems Programming Environment (SPE) is an implementation of the C library and execution framework that is designed for writing 370 systems applications. The following paragraphs provide an overview of this library and the environment it provides.

The SPE library provides support for normal programming functions, such as string-handling and I/O, and for specialized systems programming functions, such as interrupt handling and dynamic loading. The functions included in the library can be categorized into three classes. In the first class are those functions that are present in the full ANSI C library definition and that do not interact with the operating system in any way. These include the string-handling and mathematical functions. In the second class are those functions that are present in the full ANSI C library and do have operating system dependencies, such as `malloc` or `exit`. SPE versions of these functions are provided. In the third class are functions or macros that provide support for specialized systems programming functions, such as interrupt handling, or that invoke commonly used SVCs or operating system services, such as GETMAIN.

The following functions are available in the SPE library:

1. memory management, including `malloc`, `free`, and support for DMSFREE and GETMAIN
2. terminal I/O, including OS TPUT/TGET and CMS RDTERM/WRTERM
3. CMS File System and OS BSAM I/O
4. dynamic loading
5. program control
6. interrupt handling
7. diagnostic control.

Source code is provided for the functions that interface with the operating system to allow modification for specialized needs. Only functions that are required are linked into the load module to ensure minimal load module size.

The SPE version of the C environment (called an execution framework in SPE) is compact and flexible. The execution framework is created during program start-up or on entry to the first C routine. The start-up routine acquires storage for program variables, manages the stack, allows for optional error-handling, and provides for destruction of the framework and release of its resources on termination of the application. Source is provided for the start-up, exit and stack management routines. The execution framework can be managed by two basic methods.

1. The C framework can be created on entry to the main C routine and destroyed on exit. This method is heavily oriented towards traditional assembler programming concepts and best used for a main C routine with a number of subroutines.
2. Alternatively, using the INDependent compiler option, the C framework can be created on entry to the first C routine, preserved across function calls, and destroyed when no longer required. This method is best suited for a package of C service routines where no main routine will be executed.

Standard start-up routines are provided for generic cases. Since the source code is provided, the start-up and exit routines can be modified and combined with one of the two methods to conform to special environments and linkage conventions. For example, consider the following constraints that may exist for a program that runs as a CMS nucleus extension:

- The values in R0, R1, and R2 on entry to the nucleus extension must be passed to the C entry point.
- The nucleus extension may have special attributes, such as ENDCMD, which indicates the nucleus extension receives control at end-of-command processing.
- The C framework should only be created the first time the nucleus extension is created and destroyed only when the nucleus extension is dropped.

Through a combination of the INDependent compiler option and the provision for a user-written start-up routine, SPE enables C programs to run as efficient nucleus extensions. An example of

this use and one of an MVS SVC start-up routine are provided in the SPE documentation.

SPE provides source for the prologue and epilogue routines. This allows flexibility in stack management and permits the trade-off between support for debugging and good performance to be managed on an application-specific basis. For instance, an application may require that all registers be saved on function entry to facilitate dump reading or that only a minimum number be saved to improve performance. Additionally, SPE includes source for the math error-handling and for the library warning routines, providing a degree of control over the error-handling desired for a specific application.

Support

There are several additional pieces of support that can aid in using C for systems applications. One of the most important tools provided by the SAS/C compiler is the DSECT2C utility that converts assembler DSECTs to accurate, usable C structure mappings. The SAS/C compiler accepts the nonstandard extensions to C that are required by these C structure mappings. These include anonymous unions, such as those generated by overlapping fields, and noninteger bitfields.

Good documentation containing useful, real-world examples is very important and is a standard for all SAS/C features, including SPE. The SAS/C Usage Notes tape, available to any user of the SAS/C compiler, will include an example library of actual systems applications contributed by users. Vendor-provided support and maintenance are important, as the reliability of systems applications is often critical. The SAS/C compiler is known for its frequency of releases, its availability of fixes for all known bugs, and its knowledgeable, timely, and free technical support.

A source-level debugger can be a valuable tool for initial debugging of some applications and greatly enhances programmer productivity over a machine-level debugger. The SAS/C compiler includes a source-level debugger that, while it does not execute with the SPE library, can be used with the regular library during initial debugging. For instance, the debugger includes the MONITOR command to allow data objects and storage to be monitored for arbitrary changes in value. Additionally, the SPE library provides an optional post-mortem traceback facility via the `btrace` function.

BENEFITS OF USING C FOR SYSTEMS APPLICATIONS

It is clear that C is well-suited to many systems applications. While the ANSI-defined C library is not well-suited for these applications, it is certainly possible to create a C library that does support systems programming applications well. The SAS/C compiler meets this requirement with its SPE library.

In addition to the suitability of C for systems applications, there are several other good reasons for choosing C. Programmer productivity studies have shown that high-level languages require less lines of code and less coding time than assembler, while the ratio of bugs to lines of code remains approximately the same. Thus, in the hands of an experienced software engineer, it is clear that coding in C when possible is more productive. C is generally easier to maintain than assembler code. Experienced C programmers are plentiful compared to skilled assembler programmers, and C programming skills are considerably more portable across projects and hardware than assembler skills. In general, assembler programmers learn C easily. And finally, there are some systems applications for which C is clearly superior to assembler.

DECIDING WHEN TO USE C FOR SYSTEMS APPLICATIONS

The following points should be considered in any decision to use C or assembler for systems applications. C is well-suited for applications that involve parameter parsing, table searches, linked list processing, message text handling, and structured or complex decision-making. Routines of this sort written in C are also easier to debug and maintain.

Additionally, C should be considered for applications where the overhead of the C environment is not significant compared to the overall path length of the code. Code size is a lesser consideration, because in many cases C code, especially if it has been optimized, compares favorably with assembler in size and speed.

Applications that involve a very short instruction sequence or that are in extremely high-volume, critical paths are probably best written in tightly hand-optimized assembler. Additionally, exits that must run in a particular environment, such as JES2 or CICS, while possible to write in C, require a fairly sizable initial programming investment to create, support, and adhere to the environment from C.

EXAMPLES

In the paragraphs below, the broad range of systems software that has been written in C for the 370 is presented. Example applications are then discussed.

As previously mentioned, there is an increasingly large body of 370 systems software written in C. Some examples that are written using the SAS/C compiler include

- information retrieval systems
- database management systems
- performance monitors
- natural language translators
- engineering applications design tools
- source code librarians and program construction utilities.

A good example of the versatility of C is in the development of Version 6 of the SAS® System. C is used in the SAS System to perform the following functions:

- user interface and applications
- full-screen user interface
- interprocess communication
- task management
- memory management
- load module management
- I/O
- code generation.

Systems applications that are being planned at SAS Institute using the SPE library include

- a utility to install load modules into discontiguous shared segments under VM/XA™

- a replacement for an assembler TSO TRANSMIT exit that notifies users on different NJE nodes that they have mail
- an SVC 99 exit that validates the blocking factor for new data sets
- a TSO/E EXEC exit that changes the system search order for TSO commands.

CONCLUSIONS

The availability of C programmers and the support provided by the SAS/C compiler's SPE make it feasible and desirable to use C for 370 systems programming applications. Along with the productivity and maintenance benefits inherent in a high-level language, C is well-suited to many types of systems applications. Data and function pointers provide access to system data and routines. Generated code is small and fast. Structures can easily be used to describe system control blocks. Bit and pointer operators are provided.

At minimum, the ability to code machine instructions inline is needed for issuing SVCs and unusual assembler instruction sequences. Ideally, a C compiler should provide an option to generate assembler code in order to control register allocation and call existing assembler macros.

A C library for systems programming needs to be small, nonintrusive, and well-integrated with the operating system. Source code should be provided for routines that interface with the operating system or the C environment.

The installation using C for systems applications should evaluate each application in terms of suitability to C. The decision should consider such factors as:

- effect of library overhead on overall code size and speed
- nature of the programming tasks to be accomplished
- special environmental constraints
- critical nature of the application
- available programmer resources
- long-term maintenance considerations.

With wise application evaluation, C can provide considerable programming resource savings while costing little or nothing in software performance for many 370 systems programming applications. The SAS/C compiler's SPE provides a library and C environment that facilitate the development of such applications.