C in Embedded Systems and the Microcontroller World

National Semiconductor
Application Note 587
David LaVerne
March 1989

ABSTRACT

C is becoming the higher-level language of choice for microcontroller programming. Traditional usage of C depends on assembly language for the intimate interface to the hardware. A few extensions to ANSI C allow embedded systems to connect directly and simply, using a single language and avoiding detailed knowledge of the compiler and hardware connections.

HIGHER-LEVEL LANGUAGE USAGE

The desires leading to the greater use of higher-level languages in microcontrollers include increased programmer productivity, more reliable programs, and portability across hardware. Few such languages have served well when required to manipulate hardware intimately because most have been for mathematical computation. The C language has always been close to machine level. Indeed Kernighan and Ritchie[1] refer to it as not really a higher-level language; one view of C is as a higher-level syntax expressing PDP-11 assembly language.

C has gained a great deal of its reputation and popularity associated with its use for operating systems, specifically UNIX®[2] and similar systems. Many languages will do well enough for the application and utility programs of such a system, but being appropriate for the kernel indicates C can probably do the job of hardware control in an effective manner.

The needs of an embedded system, however, are not identical to the environment from which C has come. This warrants looking at C as it is and comparing it to the needs of C for the microcontroller world.

Operating Systems vs Embedded Systems

In most non-embedded programs, it is the processing which is important, and the Input/Output is only to get the data and report the results. In embedded or realtime applications, it is the Input/Output which is vital, and the processing serves only to connect inputs with outputs.

Operating systems are actually not as closely tied to the hardware as they might appear initially, and those portions which are close are not very portable. Operating systems manipulate hardware registers primarily for memory management (to map tasks), task process switching (to activate tasks), interrupt response (to field requests), and device drivers (to service requests). Because memory management hardware is so different between systems; because task process changing is so contingent on processor operations and compiler implementations; because interrupt system behavior is so varied; and because device control is so dependent on architecture and busses, these particular aspects of the operating system are not concerned with portability. As a result, they are generally kept separate, use a less convenient form of C depending on constants, and frequently are implemented in assembly language. This is not a major problem, since they comprise only a small portion of the total system, and have to change anyway each time the system is ported.

Embedded systems, by their very nature, are closely tied to the hardware throughout the system. The system consists of manipulating the hardware registers, with varying amounts of calculation and data transformations interspersed with the manipulations. As the system gets larger, the calculations may get more complex and may become a larger share of the program, but it is still the hardware operations which are the purpose of the system. Because the system in which these hardware pieces reside consists mostly of these hardware pieces, it is reasonable to hope for portability across processors or controllers for an application or product. Attempting to isolate all of the hardware operations is often impractical; using inconvenient forms of C is troublesome throughout the system and throughout its life-cycle; and implementing them in assembly language defeats the advantages of higher-level language usage and eliminates portability for those (and related) portions. For embedded systems, conveniently accessing hardware registers while doing calculations is essential.

Computer Systems vs Embedded Systems

Computational systems generally can be down the cable, and thus down the hall, from where they are used and can be whatever size is necessary to get the performance; production quantities are measured in hundreds and thousands, so price is a price/performance issue. Embedded systems end up tucked away in some of the strangest and tiniest places, so size can be a success or failure issue; quantities are often tens of thousands to millions of units, so additional chips or costs are multiplied ferociously and become a bottom-line issue.

The computer systems for which C was originally developed were relatively small and not especially sophisticated. However, as systems have grown, C and its implementation has grown right along with them. Most computer systems for which C is used now involve high-speed processors with large memory caches to huge memory spaces, backed by virtual memory. Many have large register sets. Such linear memory with heuristic accelerators allow for very large programs and fast execution. A major effort in optimization is in the allocation and usage of the registers, which tend to be general purpose and orthogonally accessible. Such systems, processor chips, and compilers compete almost exclusively in the field of speed.

Embedded systems, and most especially microcontrollers, have a different nature. While some applications may add external devices and memories to the controller, many are meant to be fully self-contained on one chip or have at most a few I/O chips. Microcontroller systems are small, are often required to fit in a physically small space, and are usually fed small amounts of power. Even when the system is externally expanded, the memories provided on-chip are significantly faster than the external memories because of buss driving. The total addressing space is usually very limited (32k, 64k) with expansion not linear. The registers in microcontrollers are usually a limited number of special pur-

HPC™ is a trademark of National Semiconductor Corporation.

RRD-B30M75/Printed in U. S. A.

pose registers, thus eliminating orthogonal usage. Speed is only one of many considerations in the microcontroller competition. Cost, package size, power consumption, memory size, number of timers, and I/O count are very important considerations.

Embedded Systems

Higher-level languages will achieve the goals of programmer productivity, program reliability, and application portability only if they fit the target environment well. If not, productivity will disappear into work-arounds and maintenance, reliability will be lost to kludges, and portability will not exist.

DESIRED TRAITS IN C FOR MICROCONTROLLERS

The environment in which C has developed is not the same as the embedded microcontroller world. What changes or extensions or implementations of C will provide the means to adapt the language? National Semiconductor Microcontroller Division has a compiler[3] developed for the 16-bit High Performance Controller (HPC™[4]) which has led to some exploration of these issues. The needs can be summarized as:

Compatibility
Direct Access to Hardware Addresses
Direct Connection to Interrupts
Optimization Considerations
Development Environment
Re-Entrancy

Compatibility

The first consideration for any such adaptation MUST be compatibility. Any attempt to create a different language, or another dialect of C, will create more problems than using C will solve. Dialects create problems in portability, maintenance, productivity, and possibly reliability. A programmer used to working in C will be tripped up by every little gotcha in a dialect; everyone will be tripped up by a different language.

Providing extensions to the language, while maintaining compatibility and not creating a new dialect, is accomplished by using the C Pre-Processor. By carefully choosing the extensions and their syntax, the use of the preprocessor's macro capability allows them to be discarded for normal C operation with non-extended compilers. By carefully choosing their semantics, the elimination of the extensions does not render the program invalid, just less effective.

Within these considerations there should be no unnecessary additions. An extension should not be made to avoid the optimizer's having to work hard. An extension should be made only to give the user an ability he would not have without it, or to tell the compiler something it cannot figure out by itself.

Direct Access to Hardware Addresses

Access to hardware addresses is improper in computation programs, is unusual in utility programs, is infrequent in operating systems, and is the raison d'etre of microcontrollers. The normal means of accessing hardware addresses in C is via constant pointers. This is adequate, if not great, when the accesses are minimal. For example

```
struct HDLC_registers
{
 ....
};
#define HDLC_1(*(struct HDLC_registers*)
  0x01a0)
```

allows reference to a structure of HDLC device registers at address 0x01a0, but never actually creates the entity of such a structure. If a debugger were asked about HDLC__1, it would not recognize the reference. If many registers and devices are involved, it becomes a problem to be handled by the programmer, not his tools. If the debugger tries to read the source for preprocessor statements, it adds significant complexity.

Another way of doing it is

```
struct HDLC_registers
{
....
};
extern struct HDLC_registers HDLC_1;
```

and providing an external file defining the address of HDLC__1, written in assembly language. This is clean, and does create the actual entity of a structure at the address, but has required an escape to assembly language for the system (although only at the system definition level). This was the first choice at National, and retains merit because the use of macros in the definition file allows the simple creation of a table exactly like the table in the hardware manual.

What is desirable, so that the user can do his own definitions without resorting to two languages, is a means to create the entities and define the addresses of those entities, a simple means of saying that this variable (or constant) is at a specific absolute address. The syntax

```
struct HDLC_registers HDLC_1 @ 0x01a0;
```

would be excellent as an official enhancement to the language, since the @ parses like the = for an initialization (and the program shouldn't initialize a hardware register this way like a variable). However, this violates the compatibility rule for an extension, since the preprocessor cannot throw away the address following the @ character. Therefore,

```
struct HDLC_registers HDLC_1 At (0x01a0);
```

is a much more practical form as an extension—and can be made to expand to the previous (or any other) form if it is ever added as an enhancement to the language. The resulting forms

```
volatile struct HDLC_registers HDLC_1 At
  (0x01a0);
volatile struct HDLC_registers HDLC_2 At
  (0x01b0);
volatile const int Input_Capture_3 At
  (0x0182);
```

are straightforward, simple, readable, and intuitively understandable, and provide the data item definitions as desired.

Direct Connection to Interrupts

Operating systems attach to interrupts in one centralized, controlled location and manage them all in that module. Embedded systems attach to varied interrupts for a variety of purposes, and frequently the different interrupt routines are in different modules with associated routines for each purpose. It is possible to do this with another escape to assembly language, but this requires that the system be maintained and enhanced in two languages.

The solution chosen for the National compiler is to provide an identifier for functions which are to service interrupts.

These functions obviously take no arguments and return no values, so they are worth considering as special. The syntax chosen was simply

```
INTERRUPT2 timer_interrupt( )
```

although a more desirable form as an official enhancement would be

```
INTERRUPT(type)
interrupt_service_routine( )
```

because the chosen syntax can be preprocessed into whatever might be the final form. The semantics of the interrupt function were more difficult to guarantee for the future—should an interrupt function be callable by the other functions? Prohibiting it allows eventually permitting it if necessary; for improved efficiency, the National compiler does not allow an interrupt function to serve as anything other than an interrupt service routine, although one function can be attached to several interrupts.

Because the functions are special purpose, the function entry and exit code can be dedicated to interrupt entry and exit, rather than having to hide it in a separate library module. The National compiler actually generates the interrupt vector to point directly to the interrupt function; the function saves and restores the registers which it may destroy. Latency is minimized.

Interrupt response speed (latency) and interrupt system performance are important characteristics of a microcontroller. It is one thing (inconvenient or embarrassing) for a multi-MIPS machine to choke on long 9600 baud transmissions and drop a character or two because of inefficient interrupt response. It is another thing entirely—lethal, a total failure—for an embedded system's interrupt response to be so poor as to miss even one critical interrupt.

Optimization Considerations

Computer systems compete on speed (or at least MIPS ratings); compilers for them must be speed demons. Microcontrollers compete on size and costs; compilers for them must be frugal. Embedded systems are limited in their memory and different memories frequently have significantly different behavior.

The major concern of optimization comes down to code size. In most controller systems, as generated code size decreases speed usually increases. The effort in the code generation and optimization should be directed towards reducing code size. Claims for exactly how close the generated code gets to hand-written assembly code depend on specific benchmarks and coding techniques. An acceptance criterion for the National HPC compiler was code size comparison on a set of test programs. A level slightly below 1.4 times larger than assembly was reached.

In addition to the implementation of the optimization, other concerns of microcontrollers affect the way code can be generated. An example is the different forms of memory. Many controllers have memories which can be accessed by faster or shorter code. Certain variables should be placed in these memories without all the variables of a module going there (which is a linker process). There is no possible way for the optimizer to guess which variables should go there, especially in a multiple module program, so it must be told. The syntax used is

```
static BASEPAGE int important_variable;
```

because the special memory in National's HPC is the first page of RAM memory. Several other possibilities offer themselves, including using for an official enhancement

```
static register int important_variable;
```

because currently static register variables are specifically prohibited. This cannot be an extension, because the register word could not be redefined to the preprocessor. If some variables need to be accessed by fast code, and some need to be accessed by short code, and if the two were mutually exclusive, it would be desirable to have two separate extension words. Since such hardware is unlikely, the single word BASEPAGE is probably sufficient.

Additional savings can be achieved by reconsidering string literals. The ANSI C requires that each string literal is a separate variable, but in actual usage they are usually constants and therefore need not be separate nor variables. The National compiler provides an invocation line switch to indicate that all string literals (but not string variables) can be kept in ROM rather than being copied to RAM on system start-up. Such strings can be merged in the ROM space to eliminate duplication of strings.

An extension to the language to identify functions which will not be used recursively is

```
NOLOCAL straight_forward_function( );
```

which causes all local variables to be converted to static variables, which are easier and faster to access and use. If the function has no arguments, the compiler can even eliminate the use and creation of the Frame Pointer for the function, saving additional code and time.

The particular processor, the HPC, has a special form of subroutine call. Since the optimizer cannot guess across modules which functions should be called with the special form, the extension

```
ACTIVE specially_called_function(arg);
```

was added. This may or may not be appropriate for other processors, but is a good example of why the language needs careful extensions to take advantage of different processors.

One command extension was added to the language because it allows the programmer to guarantee something the optimizer cannot usually determine. The form

```
switchf(value) {...}
```

provides for a switch/case statement without a default case. When speed and size become critical, the extra code required to validate the control value and process the default is highly undesirable when the user's code has already guaranteed a good value.

The National compiler has one extension which violates the issues stated under compatibility. It remains for historical reasons. It is a command

```
loop(number) {...};
```

which produces a shortened form of the for loop, without an accessible index. This does not provide the user with any new ability, it merely allows the compiler optimizer to know, without figuring out, that the index is not used inside nor outside the loop, and can therefore be a special counted form. The preprocessor cannot produce an exact semantic equivalent for the statement. This is a perfect example of a poor extension and will eventually be eliminated.

Development Environment

Languages developed for large or expensive systems can usually depend on large systems for development support, either self-hosted or with a large system host providing cross-development tools. Microcontrollers are often price sensitive, are frequently in the laboratory or the field, and are not always supported by a large system as a development host. Personal computers provide an excellent platform for the entire suite of development tools.

National Semiconductor currently provides its compiler and associated cross-development programs on the IBM PC and clone type of computer. The software is all very portable, and can be run under VAX/VMS, VAX/Ultrix, or VAX/BSD4.2, and on the NSC 32000-based Opus add-in board for the PC running UNIX V.3, and some other versions of UNIX. The demand has been for the PC version; the PC is a very good workstation environment for microcontrollers. Other environments may be desirable, but the PC is first.

Re-Entrancy

Even with all these other considerations handled, there is a time bomb lurking in C on microcontrollers. C is a single thread, synchronous language as it is usually implemented. Since most utilities are strictly single-thread and the UNIX kernel forces itself into a single-thread, this is not a big problem for them. Embedded systems involving controllers are inherently asynchronous; the language in which they are implemented must be multi-thread without special rules and exception cases.

The passing of arguments on the stack and the returning of values in registers allow for complete re-entrancy and thus asynchronous multi-threading, but this breaks down when structures are returned. Most implementations of C use a static structure to contain the returned value and actually return a pointer to it; the compiler generates the code to access the returned structure value as required. This cannot be used in a microcontroller environment, because if an interrupt occurs during the time the static structure is being used, it cannot re-enter the function. On an operating system level such conflicts can be managed with gates, semaphores, flags, or the like, but that solution is completely inappropriate on the language level. Turning the interrupts off is similarly not a language level concept, and is impossible on a system with a NonMaskable Interrupt. Telling users not to get themselves into that situation is crippling at best, impossible to enforce, and extremely difficult to track down and correct.

The solution should be at the language level, and should allow the return of a structure without hindering re-entrancy. The author's solution, developed with National, has been to have the code calling the function provide the address of a structure in which to build the return value. Since this is frequently on the caller's stack, and is never invisibly static, the program has no hidden re-entrancy flaws.

The HPC C Compiler

The HPC C Compiler (CCHPC) is a full and complete implementation of ANSI Draft Standard C (Feb 1986) for free-standing environment. Certain additions take advantage of special features of the HPC (for the specific needs of microcontrollers). The extensions include the support of two non-standard statement types (loop and switchf), non-standard storage class modifiers and the ability to include assembly code in-line. The compiler supports enumerated types, passing of structures by value, functions returning structures, function prototyping and argument checking.

Symbol Names, both internal and external, are 32 characters. Numerics are 16-bit for **short** or **int**, 32-bit for **long**, and 8-bit for **char**, all as either **signed** or **unsigned**; floating point are offered as **float** of **double**, both using 32-bit IEEE format.

All data types, storage classes and modifiers are supported. All operators are supported, and anachronisms have been eliminated (as per the standard). Structure assignment, structure arguments, and structure functions are supported. Forward reference functions and argument type checking are supported.

Assembly code may be embedded within C programs between special delimiters.

See Table I.

CCHPC SPECIFICATIONS

TABLE I

Note: Extensions are boldface

Name length		32 letters, 2 cases
Numbers		
Integer,	Signed and Unsigned	16–32 Bits
	Short and Long	16 bits and 32 bits
Floating,	Single and Double	32 bits and 32 bits
Data Types		
Arrays		
Strings		
Pointers		
Structures		

Preprocessor

#include
#define #define( ) #undef
#if #ifdef #ifndef #if defined #else #elif #endif

Declarations

auto register const volatile **BASEPAGE**
static static global static function **NOLOCAL INTERRUPTn ACTIVE**
extern extern global extern function
char short int long signed unsigned float double void
struct union bit field enum
pointer to array of function returning
type cast typedef initialization

Statments

;{...} expression; assignment; structure assignments;
while ()...; do...while (); for(;;;)...; **loop( )...;**
if ()...else...; switch ()...; case:...; default:...; **switchf ( )**...;
return; break; continue; goto...; ...:

Operators

primary: function( ) array[] struct_union. struct_pointer ->
unary * & + – ! ~ ++ -- sizeof (typecast)
arithmetic: * / % + – << >>
relational: < > <= >= == !=
boolean: & ^ | && ||
assignment: = += –= *= /= %= >>= <<= &= ^= |=
misc.: ?: ,

Functions

arguments: Numbers, Pointers, Structures
return values: Numbers, Pointers, Structures
forward reference (argument checking)

Library Definition **Limited-Freestanding environment**

Embedded Assembly Code

CONCLUSIONS

With the right extensions, the right implementations, and the right development environment, National is providing its customers with a C compiler tool which allows effective higher-level language work within the restrictive requirements of embedded microcontrollers. Productivity increases do not have to come at the expense of larger programs and more memory chips. No strangeness has been added to the language to cause reliability problems. Portability has been retained. Assembly language code has been eliminated as the chewing gum and baling wire trying to hold it all together, further increasing reliability and portability.

FOOTNOTES

1. Kernighan, Brian W. and Ritchie, Dennis M., ''*The C Programming Language*'', Prentice-Hall 1978, Pages ix and 1.
2. UNIX® is a registered trademark of AT&T.
3. Produced by Bit Slice Software, Waterloo, Ontario, Canada.

ADDITIONAL INFORMATION

Datasheet
HPC Software Support Package

User's Manual
HPC C Compiler Users Manual #424410883-001

Lit. #100587

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-8090
Tel: 1(800) 272-9959
TWX: (910) 339-9240

National Semiconductor GmbH
Livry-Gargan-Str. 10
D-82256 Fürstenfeldbruck
Germany
Tel: (81-41) 35-0
Telex: 527649
Fax: (81-41) 35-1

National Semiconductor Japan Ltd.
Sumitomo Chemical
Engineering Center
Bldg. 7F
1-7-1, Nakase, Mihama-Ku
Chiba-City,
Ciba Prefecture 261
Tel: (043) 299-2300
Fax: (043) 299-2500

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductores Do Brazil Ltda.
Rue Deputado Lacorda Franco
120-3A
Sao Paulo-SP
Brazil 05418-000
Tel: (55-11) 212-5066
Telex: 391-1131931 NSBR BR
Fax: (55-11) 212-1181

National Semiconductor (Australia) Pty, Ltd.
Building 16
Business Park Drive
Monash Business Park
Nottinghill, Melbourne
Victoria 3168 Australia
Tel: (3) 558-9999
Fax: (3) 558-9998