C language support for small scale embedded systems

JTC1/SC22/WG14 N924

Walter Banks
Byte Craft Limited
421 King Street North
Waterloo, Ontario N2J 4E4

Voice : (519) 888 6911
Fax : (519) 746 6751
Email : walter@bytecraft.com

1. Introduction:

These comments reflect the needs of small scale embedded developers. The effective expression of an application in a high level language requires appropriate vocabulary and syntax, and language that grows with the industry that embraces it.

Our industry is growing up from the cottage industry to a multi-billion dollar industry. Software is an industry of individuals implementing everything in an individual way. We are now using high level languages to implement applications small scale embedded systems.

Small scale embedded systems are currently supported by many C compiler vendors each offering their variation on the C language loosely based on various ISO/ANSI C language standards. None of these C language development tools come close to any specific recognized standard. Most of the C compiler vendors would like a standard to benchmark their products against, and their customers would like standard features implemented in a standard way. The existing standards fail to provide reasonable working definitions of the C language that can be effectively implemented on the target processors used by small scale embedded systems. The C compiler vendors for small scale embedded systems have each added new features to the language and structured their implementation around the needs of their customers and the constraints of the runtime environment. The lack of high-level language standards has resulted in babble speak and inefficient application implementation.

C standards for embedded systems are needed to meet the communication needs between the application developer and the application. The C language has the dual purpose of providing an abstract description of the target environments and as a portable

description of an algorithm that can be moved from application to application and processor to processor.

We have tended to believe that small low cost embedded systems are one-of-a-kind applications while failing to recognize that as many as 90% of all low end embedded system applications could be implemented on at least one member of the leading dozen of so microcomputers. The small scale embedded systems implemented in a high level language (most often a C dialect) are now being regularly employed in watches, microwave ovens, bathroom scales, children's toys, personal organizers, TV remote controls, and automotive applications. Embedded small-scale products have become very complex. Code quality from many compilers rivals that from the best hand coded assembly language. In other words, we as an industry can benefit from standardized features.

The traditional view of low-cost embedded microcomputers is that they have many different types of non-standard I/O hardware. By volume, as many as 50% of the embedded microcomputers are sold to the volume manufactures for less than $1 US. The view of application developers was that each vendor offered a particular innovative solution to embedded systems. Part of the application development process has been to match the facility offered by the silicon vendor to requirements of the application. Of course this has to some extent been true, however competitive pressures over the last twenty years have created a feature similarity between many competitive offerings.

Applications that are developed for low cost embedded systems are not particularly sensitive to development cost. Applications themselves are innovative and unique. So what is the incentive for standardization. Time to market, reliable reusable code, product reliability and product liability. Standardization will focus the attention of the third party software vendors to develop portable applications and libraries. Twenty-five years has brought many similarities between most of the popular processors. A study we did found remarkably few variations in function, only in silicon implementations

The major high level language problem in small scale micro-controllers are the need for low level library routines, the standard description of multiple memory spaces in applications, and a consistent, portable method of accessing differing numerical representations.

2. Rationale

Any standard that will be widely accepted and used for code generation on small-embedded system processors will need to address the following issues.

2.1 Diverse memory support.

Embedded systems have many different forms of memory, and the languages that support them need to be flexible enough to address memory related issues like the following.

2.1.1 Multiple address space support.

The C *register* address space modifier recognizes that all memory is not the same and the programmer has the option specifying that some variables can

be assigned to one or another memory form. All embedded systems support multiple address spaces (even the Motorola 6805/6808/6811 and the 6502). This is not the traditional Von Neuman vs. Harvard architecture argument but also includes issues of first page access Vs all of RAM space and multiple non contiguous memory area's, I/O address space. There are processors that have separate RAM and ROM address space that in some configurations can directly access ROM including writing to it.

2.1.2 Device driver based memory support.

Many embedded systems include memory that can only be accessed with some form of device drivers. For example, serial data busses including I2C, SPI, and on board flash memory in processors used as a form of non-volatile memory. Device driver memory support is needed in any application where the details of the access method needs to separated from the details of the application.

Device driver based memory support allows applications to utilize symbol table handling and allocation management features of a compiler. It has been found that the compiler optimizer is easily capable of making calls to the appropriate device drivers whenever a reference is made that requires calls to the user supplied device drivers. The compiler implementation issues are similar to processors requiring memory management support for data access. There is an argument that device driver based memory support could be handled under C++. At this time there is very little movement for small scale embedded systems to be implemented using C++.

2.1.3 I/O device definition.

I/O registers are a unique special form of memory accessed directly with specialized instructions or are mapped on one of the memory areas. Most processors consistently address I/O registers with only minor variations between different parts. The compiler needs information to understand I/O register variations and the data structure and address of each I/O device. Access methods include read/write, read only, write only and read modify write. Operations on I/O devices are part architecture of the processor and within a processor family can be safely be encapsulated within the compiler.

2.2 A rich selection data types and math support.

Embedded developers want rich resources when they express themselves through applications. Developers see the need for processor-based support where the *int* of one processor fits the needs of the processor but may not be the same size or characteristic as the *int* from another processor.

Developers also want to declare variables of a specific size, independent of the platform on which they are to be implemented. This supports two forms of portability that dominates the embedded system world. Algorithms that are ported between applications may have some data specific requirements and data structures transmitted between separate embedded system applications require very specific data sizes in order to maintain compatibility.

2.3 Embedded standard input output libraries.

ISO/ANSI C standard libraries were developed primarily for much larger systems either hosted applications or large, low volume embedded systems. These libraries accomplished a lot for making the developed code very portable. Development systems and application developers became more productive. Code for critical and often difficult I/O functions is now stable, reusable, and well understood.

Standardized embedded system libraries are needed to support the common I/O devices relevant to embedded applications. The goal is to define an embedded *stdio* that provides common calling conventions for parallel ports, serial communication, serial bus support, watchdog and timer management. The emphasis is on portability between target platforms.

3 Implementation details

3.1 Diverse memory support

3.1.1 Multiple address space support.

The traditional C approach to multiple address spaces is handled with modifiers. C provides for the *register* modifier. Most of the compiler vendors have extended this concept to provide support for additional memory address spaces as dictated by the target environment.

It may be reasonable to allow compiler-specific modifiers to be used and the actual placement of data in the multiple address space environments to be resolved by the linker. The format of the allowable address space modifiers should be such as to reduce or eliminate variable naming conflicts. One possibility is to incorporate a fixed prefix for example *_memory* into the modifier name. The special function register space found on many processors might then be referenced as *_memory_sfr*.

The biggest C implication of multiple address space support is mapping pointers to each of the address spaces. The pointer problem can be briefly described as how is,”memory de-referenced when pointers are passed in an application with multiple address spaces?”

3.1.2 Device driver memory support.

Many small-scale embedded systems applications include the use of memory that can only be accessed with some form of device drivers. For example, serial data busses including I2C, SPI, and on board flash memory in processors used as a form of non-volatile memory.

Extending the multiple address space modifier system just described for use with user defined memory requires a formal link in the compiler to be established between the variable allocated in the address space and drivers that can read and write that data. In a test implementation, data allocated in _m*emory_my_mem* assumed that functions would exist to resolve calls to
char memorymy_mem_r(address)
and
void memory_my_mem_w (address,data)

These functions were called directly as part of the code generation of the compiler. The function names were constructed from the symbol table information for the variable. The linker resolves the calls to the user supplied function definitions.

3.1.3 Input/Output device definition.

I/O registers are a unique special form of memory accessed directly with specialized instructions or mapped on one of the memory areas. Most small scale embedded systems processors address Input / Output registers in the same manner with only minor variations between different parts. The information that the compiler needs to understand these variations and the data structure and address of each I/O device is:

3.1.3.1 Permitted access methods: read/write, read only, write only and read/modify/write. These definitions should be encapsulated within a modifier and used by the compiler for usage consistency checking.

3.1.3.2 Physical address. Many of the current embedded C compiler vendors have adopted the @ operator to define a physical address rather than a pointer to a constant. Both forms could be used however it has been found that the @ operator is easier to understand.

It is important to note that, in architectures with banked memory spaces, that one physical port may appear at several different addresses. The syntax used for port definitions should reflect this.

3.1.3.3 Port data structure. Many small-scale embedded systems hardware ports include a data structure with several packed data fields. The current C support for structures and will serve to define the individual fields within a port.

Operations on I/O devices are part of the architecture of the processor and within a processor family can be safely be encapsulated within the compiler.

Some examples of port declarations.

The following declaration of a port with read and write permissions located at address 0x90 of the *_memory_sfr* address space.

```
_memory_portrw char port1 @ _memory_sfr 0x90;
```

The status register of a serial port may, for example, be defined as the following structure. Bits may be individually accessed at a very low level directly from C. It should also be noted that the port in this case is declared as read-only. Writes to any of the referenced bits should produce compiler warnings.

```
_memory_portr struct
    {
      int tdre      : 1;
      int tcie      : 1;
      int rie       : 1;
      int ilie      : 1;
      int te        : 1;
      int re        : 1;
      int dummy     : 2;
     }
    sccr2 @ 0x000F;
```

Data fields may also be accommodated as the following declaration shows. This is also a case of read/write access. The dummy variables are placeholders in the blank fields. In both of these examples the port has been mapped on the normal address space.

```
_memory_portrw struct
    {
      int dummy1        : 2;
      int scp       : 2;
      int dummy2        : 1;
      int scr       : 3;
     }
    baud @ 0x000D;
```

3.2 A rich selection data types and math support.

The C language needs to address two data typing needs:

1. The common data types that are target implementation specific where int's are the natural size of the processor or 16 bits and the remainder of the standard data names reflect the underlying architecture of the target environment. These data types are handled with the current char, int, long and double combinations.
2. A clean method of the application to specify specific data types. Portability is usually cited but is certainly not the only valid reason to specify a specific data type for a variable.

The needs of small scale embedded systems requires the following data types

3.2.1 Integer support for 8*2^n types of standard 8,16,32,64 ... bit data.

3.2.2 Integer support for data sizes that are not 8*2^n (for example 24 bits which are the more rational choice for many embedded applications)

3.2.3 Definition in a standard way for data types of a specific size. (stdint..h)

3.2.4 Support for fixed point data types in addition to int and float types. There has been a lot of formal justification in two of the earlier papers. The widely circulated N854 DSP-C paper describes, fixed point requirements from a DSP perspective in a lot of detail. Willem Wakker has refined this paper in N907, and taken a position on fixed point with which I largely agree.
Fixed-point data types should be defined as a combination of integer and fraction part for example fixed16_8_t where the integer part is 16bits and the fraction is 8 bits. This is a superset of the N854 description and requirements of fixed-point data types.

3.2.5 Support for the *saturated* modifier. Section 2.3 of the N907 paper makes a case for a saturation modifier. I agree with Mr. Wakker's comments on this point. The saturation modifier says that a variable declared with the saturation modifier in arithmetic operations will saturate at maximum positive or negative values rather than wrap as a result of math operations.

3.3 Embedded standard input / output libraries.

The standard C libraries fail to address the needs of small scale embedded systems. In a survey of most of the available processors a remarkable amount of similarity between the offerings of each of the embedded 8-bit processor families. Together these processor families have in excess of 95% of the market in both application designs and production volume.

Library calling sequences are based on the ability of a library function to be implemented on a broad spectrum of common embedded systems. The library calls themselves should be application oriented and not hardware implementation specific. A Digital to Analog output should be transparent to the application

developer, whether it is implemented with a D to A converter on a parallel port or implemented with a PWM. Efficient compilers will compensate for potential losses from standardized calling sequences in standardized libraries. There are a number of ways that compilers in the next decade will be able to compensate for standardized libraries. The tradeoffs are comparing specialized functions or inline code with the overhead for standardization libraries. I have taken a fair amount of space in this paper to describe both a measurement and a methodology that will allow code developed around embedded system libraries to be effective in an application.

The following library function definitions illustrate some of the potential candidates for inclusion into a standard embedded system library.

3.3.1 Parallel Ports

Port support is perhaps the simplest of the I/O support for embedded systems. The major complicating factor for port support is initialization of the data direction control registers. On some processors this requires a one and others a control bit of zero. Our proposal is a common call for initialization of the port for output. This call will assume that the port is otherwise configured for inputs.

void initPORTwrite(port,output);

Initialize the port for outputs. A one means the corresponding bit is an output. The call assumes that the remaining bits are always inputs.

```
void   writePORT(port,data);     // Write a byte to an output port.
char   readPORT(port);           // Read a byte from a port.
```

Although reads and writes to a port may be achieved through function calls, it is more likely that these calls are realized with inline code generated by the compiler. Port accesses within a processor family have been found to be consistent and it is reasonable to establish access validation and restrictions within the code generator of the compiler.

3.3.2 SERIAL I/0

The serial port I/O library is perhaps more complicated that it needs to be. It allows for support for both synchronous and asynchronous serial ports. There is support for more than one serial port as well. The library design is organized to interface to the ANSI C library functions putc and getc through simple macro calls. The serial ports may be implemented either through serial port hardware or through software-implemented UART's.

```
void initSerialPort(port,sync/async,baudrate,databits,parity,stopbits)
void _putc(port,ch);
char _getc(port);
```

3.3.3 MICROWIRE I2C SPI

This family of I/O support buses can provide a wide variety of I/O support facilities ranging from inter-processor communications to interfaces with external serial and parallel ports, RAM, ROM, EEPROM, A/D and D/A devices.

```
void   initSPI(port,config);
char   rwSPI(port,data);
char   readSPI(port);
void   writeSPI(port,data);
```

3.3.4 Analog to Digital

Embedded microcomputers have between one and eight analog input channels, with current resolution between 8 and 16 bits. Most of the A to D converters need some form of general setup. Typical setups select a reference source and sometimes resolution and conversion time parameters. In looking at a lot of application code this is never changed over the course of code execution in an application. The two supporting calls follow. The initialization is a single call loading configuration information. The readAtoD call will read the A to D value on a specific channel.

```
void   initAtoD(void);
int    readAtoD(channel);
```

3.3.5 Digital to Analog

Very few embedded systems have a D to A converter built into the into them. In some applications, Pulse Width Modulation (PWM) ports are used to generate an analog output voltage. These library calls are implementation independent and can be used effectively even when support for the PWM also is being used. The support calls for the D to A follows.

```
void   DtoA(void);
void   writeDtoA(analogvalue);
```

3.3.6 Pulse Width Modulation

Pulse Width Modulation (PWM) ports are very flexible output ports that can generate levels, sophisticated pulse trains, and with the addition of a simple low pass filter they can generate analog outputs. The number of PWM channels varies among 2 and 16, and the resolution of the generated pulse stream is among 6 and 14 bits.

```
void   initPWM(port);
void   PWM(port,frequency,dutycycle);
```

3.3.7 Delays

Library routines that will impliment a delay in real time for the application. The standard delay times are in seconds, milliseconds or microseconds. The implimentation can use hardware timer registers or software delay loops. The user interface is:

void delay(time_seconds);
void delay_ms(time_milliseconds);
void delay_us(time_microseconds);

3.3.8 Watchdog Timers

Watch dog timers are used by embedded systems as a check against runaway execution of code. The hardware implementation of watchdog timers varies considerably between different processors. In general, Watch Dog Timers must be turned on once, often within the first few cycles after reset, and then reset periodically within the software. Some of the watchdog timers have the ability to be programmed for different time-out delays. The reset sequence is sometimes as simple as a specialized instruction or as complex as sending a sequence of bytes to a port. Watchdog timers either reset the processor of execute an interrupt when they time out.

The following are the proposed library functions supporting watchdog timers.

void initWD(timoutdelay);

Watchdog timer initialization call. “timeoutdelay” is maximum time between clearing the watchdog timer.

void clearWDT0(void);
void clearWDT1(void);
void clearWDT(void);

This call will clear the watchdog timer. Some processors support a sequence that needs to be called in sequence to clear the watchdogtimer. The first call is clearWDT0 followed by clearWDT1. The call clearWDT will clear the watch dog timer it will generate the complete sequence if necessary to clear the watchdog timer.

4 Conclusions

What will these changes to the non-hosted C standards achieve? It will bring a consistency over time for the code creation products for small scale embedded systems.

The incentive to adopt and support the standards will be high if the standards support the requirements of this part of the industry. Increasingly, it will provide even more portability between widely different target micros. The role of the chip vendor will be to supply silicon in an ever-increasing competitive marketplace. This will reduce company loyalty. It will also provide an incentive for the third party software vendors to provide compatible cross platform tools.

There are benefits for software reliability in using standardized libraries. The first benefit is dramatically improved system reliability, stemming from a well-defined user interface; this reduces the number of series terms in the reliability equations. Secondly reduced development cost through the use of well developed and (I assume) well-debugged standardized modules.

Standardized calling sequences will improve the overall reliability of application code. Reusable code reduces application development time, minimizes debugging, improves product reliability thereby reducing potential liability suits.

Acknowledgments

Andre Labelle and Sherif Abdel-Kader implemented the embedded systems libraries on the Microchip PIC, Motorola C6808, Scenix SX, National COP8, 8051, and Cypress USB M8 series. Bruno Bratti was responsible for surveying the available I/O facilities on most of the popular microcomputer products. The work of these people at Byte Craft contributed to much of the material presented here.

I would also like to thank Jan Kristoffersen for his comments on a draft of this paper.