C Language support for multiprocessor application environments.

Walter Banks
Byte Craft Limited
Canada

February 2009

The purpose of this paper is to start the dialog that must inevitably precede making a clear decision on the emerging issue of support for multiprocessor applications. This paper is meant to define the problem, its terms and current solutions and some of the possible alternatives. This paper is not proposing a specific C language solution to language support for multiprocessor systems. There may not be a single language solution, and at this point support technology may still be viewed as not fully mature even though much of it is 25 plus years old.

The following two questions should be considered by WG-14 regarding the support of multiprocessor execution environments.

1) Should the mainstream language support a multiprocessor environment? **(Currently N1256 Programming languages — C ISO/IEC 9899:TC3)**

2) A separate but related issue is should the embedded systems technical report be enhanced to support named execution spaces to reflect current embedded systems practices? **(ISO/IEC TR 18037 Programming languages - C - Extensions to support embedded processors)**

There may be other documents that could be affected and they may need changes as well.

Multiprocessor environments date back 40 years to the I/O controllers of large mainframe computers. There have been many attempts to measure and define how multiprocessors should be defined and evaluated.

In the mid 60's Flynn attempted to define multiprocessor systems in terms of data flow and processor simultaneous or independent execution. Flynn's definitions were all combination of single or multiple instruction streams and single or multiple data streams. (SISD SIMD MISD and MIMD) Flynn's definitions went through many modifications to attempt to classify multiprocessor implementations. In time it became clear that many of the early applications for multiprocessor environments represented special but important

cases. The first proposed parallel applications were all variations on matrix calculations weather forecasting, pattern matching specifically Grace Hopper's ground data cruise missile navigation pattern matching and high-energy explosion simulation.

Once the small list of obvious applications had been implemented the focus shifted to the holy grail of multiple processor support. The Holy Grail is a single application program automatically being distributed over multiple processors. Multiprocessor embedded systems generally have several specialized processing elements that can be part but not the whole solution. Many applications may never benefit from multiprocessor implementations because no significant opportunity for parallel execution. It doesn't mean that a large application may not be broken down into small functions to function on a distributed network of resources. (Indeed there are many cases where this is currently happening)

The ad hoc applications have proven to be much more the norm than the exception. Incrementally these applications have significantly contributed to the multiprocessor technology.

The following are a few examples from personal experience that illustrate the point.

The four-processor computer network simulator used for computer network protocol simulation used independent processors that ran independently but interfaced through well-defined protocols. This simulator used both common data space and messaging interprocessor communication.

This system could be described as implemented as a loosely coupled multiprocessor. Each processor ran code that could have been replaced with code that interacted with the interprocessor protocols. The C implementation requited no language extensions and no multiprocessor specific support.

This project defined the nature of multiprocessor systems. Processor coupling could be tight or loose. Tight coupling was an environment where execution threads could pass from one processor to the next and return to the original processor. Tight coupling required that execution protocols be able to call functions in alternative processors a change that would have required changes to the descriptive language used for implementations.

Loose-coupled processors function independently and interact through data passed between processors but not execution threads. Another way of defining loose coupling is each processor interfaces through a well defined protocol allowing individual processors or their application code to be replaced by with independent code that conforms to the protocols.

The mid 70's brought a lot of research on the interconnection between multiprocessor systems, not unlike some of the current debate on on-chip interprocessor connections. Academic institutions have written papers on every one, two and three-dimensional

topology. At the application level how data is shared and transported has had little impact on software implementation.

Loosely coupled processors are currently the most common form of multiprocessors. Desktop personal computer keyboards, automotive displays, distributed intelligence using CAN, LIN and other packet switching busses for data communications. Loose but close coupling with processor arithmetic co-processors.

Loosely coupled systems typically are coupled in data space, either real or virtual, that is accessible from both processors. One characteristic of a loosely coupled system is that the data life in the common communication data space is often long, rarely changing after initialization during start-up and (relatively) rarely changing during application execution. The software design in each of the processors can be independently developed as long as both parties understand the meaning of the data and its access protocols.

Another way to distinguish multiprocessor environments would be to determine if the actual application crosses between boundaries. These applications can be loosely or tightly coupled and is independent of interprocessor communication technology. The relationship between a keyboard embedded processor and the host processor in a desktop personal computer is one way of looking at this distinction. It is clear that the keyboard is part of many applications but it is very rare that the code running in the keyboard is part of a specific application. This simple distinction alters the way many applications are viewed.

The most common embedded systems solution for control of automotive engines is a multiple processor solution implemented as a hiarchity of execution platforms. A variation on a Power PC is used to provide over all engine control, it is coupled to processors that interface directly to the actual engine. This lower layer has two identical event driven processors that respond system events. Events can originate from timers, sensors or be host initiated. The division of labor between the two processor levels that contribute to the successful execution of the application and the division of the engineering skills needed to implement the application roughly follow the same lines.

Processes that are often engaged in interactive execution characterize tightly coupled systems. Data life usually is short, generally limited to a few instruction times. Response times to events requiring both processors are generally slower than loosely coupled systems.

The distinction between loosely and tightly coupled systems is really a software design distinction. The loosely coupled system has independent pieces of software that conform to the protocol standards of the communication area. Tightly coupled systems require software to have intimate details of code running in both processors and are likely to have software developed by a single software team.

This software design difference can actually be exploited by the development tool sets to reduce development time and automate the interface protocol enforcement in the communication link by the tools.

Multiple software teams divided roughly along the divisions between processors implement many loosely coupled systems. The development teams implementing the software work quite independently with independent schedules.

The development teams function independently and, although they have agreed on an interprocessor protocol, there is nothing to prevent minor changes from creeping into data structure. This is a significant problem that cannot be solved reliably by programmer discipline.

The engine controller project automated the checking of the interprocessor interface compatibility without seriously impacting the development process. The solution turned out to be surprisingly simple. The development teams would agree on communication protocols and common data memory contents. The build process for the multiple processor system would compile and link one processor and export the interprocessor interface to the second processor's build process in the form of a header file. If the interface was compatible the build would succeed; if not, then a build or compiler error would be generated.

The first processor team ultimately made the actual choices on the placement of variables and do the actual maintenance of the interprocessor space. Application design documents detailed the interprocessor space contents and common variable name and types. At make time the normal diagnostics of the compilers would identify missing variables and type mismatches.

This simple approach solved many workflow problems for multiprocessor applications. It identified intera-team and design communication errors. In normal operation it was an invisible solution.

Teams rarely develop in sync. The multiprocessor build process meant that each team could release software revisions independently of the other. Multi-versions of each piece could reasonably be expected to work together.

The implementation required tools to be able to export information in files drawn from data that normally was retained within the compiler and linker tools. Interestingly, we actually only needed to export from the tools for one of the processors. A pragma based report generator was implemented with full access to the compiler symbol table, code image and compiler resource management information.

The report generator pragmas were part of the application source and could be distributed through the code or not as a style option. The report generator ran at the end of the compile/link phase of source translation.

Case for named execution space.

Handling single chip processors with multiple execution units.

In the embedded systems world this is actually quite common. A single chip processor is created with multiple execution units. These execution units are usually heterogeneous often with diverse architecture and instructions sets. Communication between processors is typically through shared memory but may be also through other forms. The internal complexity ranges from a host processor with a programmable I/O controller to true co-processors. Single chip processors configured this way have been able for about 20 years. Some examples include Zilog 86C94 used in disk drives. Motorola 68K/TPU automotive and industrial controllers, Freescale 68HC12X/XGATE xgate is a general purpose RISC processor used as a true co-processor. 68S08/xgate used as special purpose processor primarily in automotive. Freescale PowerPC / eTPU and Coldfire /eTPU used in industrial controllers and automotive engine controllers. Not identified are all the processors with protocol co-processor that interfaces with standard bus protocols and processors that are available for executing application code.

All of the recent examples have parts where the execution space is in some way connected

These processors are easily configured to support applications that are being tightly or loosely coupled with code that is implemented for a single application or as parallel separate applications.

Consumer electronics multiple processor environments

Many high volume consumer goods manufacturers have a unique development work flow. The whole application is prototyped in a single processor. This allows them to quickly produce many working prototypes for product evaluation even small run product for marketing tests.

The production engineering phase goes though a cycle of cost reduction of the final product. During this phase all the usual stuff happens and very often this single processor is replaced with several processors. The three main reasons that this is done is to reduce overall production bill of materials cost, to reduce assembly costs by replacing wiring bundles with two or three line communication links between functional units and to run the processor clocks at a lower speed to enable the production product to meet FCC and other countries RF radiation requirements.

The original application software is functionally separated and divided among the multiple processors. The application software remains surprisingly intact with the addition of interprocessor communication functions to pass data and request interprocessor services.

The application dividing process of distributing the single processor application code among multiple processors is an interesting one. Independent of whether it is automated or hand divided each processor becomes a geographical center of reference so that software that is associated with the I/O devices of that processor becomes attached to that processor and where possible the next layer of calling software gets located in the same processor that contains lower level called functions. In a similar way data is distributed among processors based on where it is referenced and on available space.

Off processor data references are easily handled using IEC/ISO 18037 user defined data spaces. Each off processor reference is handled through an application set of data access primitives. Named execution space or user defined execution space would be a logical extension to IEC/ISO 18037 to support this type of development.

Interprocessor calls

Interprocessor calls in a multiprocessor application environment need to be handled separately. In a single processor environment calls and code execution is executed as a sequential process. In a multiple processor environment calls initiate execution in a second processor but what happens to the first processor? It can continue on (non blocked) or it can wait for execution to complete in the second processor and then continue (blocking)

There are alternative approaches to handling interprocessor calls in a multiprocessor environment but we treated all void functions as initiating an action in a second processor and immediately continuing with execution (non blocking) and any function that returns a value was implemented as a blocking call. This simple approach is not perfect but is easy to understand and visualize. The most common missed case with this heuristic is that of initiating a non-blocking off processor call that is expected to return values in the future. In our implementation this was done by returns though global variables with a void function. The data was protected through of semaphores.

Addition to IEC/ISO 18037

The named address space in the two examples cited was implemented using pragma's. WG-14 should consider adding named execution space to IEC/ISO 18037. The addition of execution space is consistent with some current embedded systems applications. The additions would have limited impact with fundamental definitions for named execution space to section 5 and a section of reference material in Annex B should discuss design considerations for off processor calls and flow control in applications using multiple processors.

Alternative implementations and how others are doing this

IEC61131 / IEC 61499 Approach. This set of standards primarily used in programmable logic controllers (PLC). IEC 61499 grew out of the earlier IEC 61131 is a system of function block programming which focuses on tracking data and system control and hiding the block implementation. This approach focuses on a programmer's comprehensive look at the application.

This last observation is an important part of multiprocessor support. It is necessary to be able to tie the complete application together, not just by design but also in someway validated by the development system as a whole. This comprehensive application view is lost when individual processors are programmed with individual application code rather than a single application wide application source.

VHDL and several C like languages have been used to program FPGAs and CPLDs at a very low level. These are implementation approaches that follow the data and control at a lower level. The compiler tools attempts to extract and exploit parallelism at a low level. This approach would change the fundamental nature of C as a language.

Actions for WG-14 to consider

ISO/IEC TR 18037 proposed change.

Add execution unit space to named and user defined address space currently support in the ”Programming languages - C - Extensions to support embedded processors “ ISO/IEC TR 18037. This extension would be relatively minor. This change does conflict with the much larger issue of supporting C language multiprocessing support should be handled.

C language support for multiprocessor programming

This paper is not taking a position on the C language changes that could be added to support multiprocessor environments. The author feels strongly that WG-14 should have specific reasons why multiprocessor language support is included or excluded in the C standards documents.

1. We can do nothing and take the narrow view that WG-14 is to considered language support for single processors. Even this may require us to define the scope of C as a language and the supporting execution environment. Part of this position may be that multiprocessor general purpose execution environments are application specialized and C is primarily an implementation language not an application language.

2. An argument for C to support multiprocessor environments is to define the language as having a full application view. If a single application is distributed across multiple processors and described as having one C source then multiprocessors should be supported.

3. We could consider the larger possibility of defining the behaviour and language support for multiprocessor environments in the context of C language support. This paper generally is not comprehensive. There are open problems that would need to be studied and addressed if multiprocessor support is to become part of the C language in normal C hosted environments. ISO/IEC TR 18037 deals with some of the data address space issues. .

    i. Off processor data references
    ii. Off processor function references
    iii. Defining off processor function calls
    iv. C may need to address the larger issue of event driven function execution. There are a few processors that have no ability to have a ***main*** function.