WG14 N1386

Additions to ISO/IEC TR 18037 to support named execution space.

Walter Banks
Byte Craft Limited
Canada

April 2009

Named execution addition to IEC/ISO 18037

This paper outlines additions to **N1275 ISO/IEC TR 18037 Programming languages - C - Extensions to support embedded processors** to add named execution support. The addition of named 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.

Rationale

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 take other forms instead or in addition. 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 in production for about 20 years. Some examples include the Zilog 86C94 used in disk drives. Motorola 68K/TPU were offered as automotive and industrial controllers. Freescale 68HC12X/XGATE contains a general purpose RISC processor used as a true co-processor. 68S08/XGATE is used as a special purpose processor primarily in automotive applications. Freescale PowerPC/eTPU and ColdFire/eTPU are used in industrial controllers and automotive engine controllers. Texas Instruments TMS570 Dual Cortex R4 is used in highly reliable industrial controllers.

It should be noted that processor combinations are both heterogeneous and homogenous.

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 tightly or loosely coupled, with code that is implemented for a single application or as separate parallel applications.

Multiple processor environments in consumer electronics

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 and even a small product run 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 are: 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, 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. Software that is associated with the I/O devices of a 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-specific 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 are 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-blocking) or it can wait for execution to complete in the second processor and then continue (blocking). The behavior of the calling processor after initiating a non-local function call is implementation defined. In some implementations it may be desirable for this to be application defined.

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 using semaphores

Additions to N1275 ISO/IEC TR 18037

The following additions to 18037 are needed to support named execution space:

> ***5.4 Named execution space***
>
> Named execution space adds an optional execution-class function specifier to function declarations. The purpose of the execution-class modifier is to connect a C function implementation to a specific execution unit.
>
> Each of the execution-class modifiers is a unique name in the form of an identifier associated with a processor execution unit. Execution-class modifiers are ordinary identifiers, sharing the same name space as variables and typedef names. Any such names follow the same rules for scope as other ordinary identifiers.
>
> An implementation will provide an implementation-defined set of *intrinsic* processor names that are, in effect, predefined at the start of every translation unit. The names of intrinsic processor names must be reserved identifiers (beginning with an underscore and an uppercase letter or with two underscores). An implementation may also optionally support a means for new processor name to be defined within a translation unit.
>
> The execution-class modifier name must be initially declared as follows; then it may be referenced as a part of a function declaration.

__Processor ***execution_class_name*{[*n*]} = *processor_id*{,*optional arguments*};**

Execution class names may be either a simple name or an array declaration associated with a single processor_id. An execution class array defines several processors that share identical characteristic that are individually distinguished by the array index.

Optional arguments are implementation defined. They provide processor- or compiler-specific information with an execution class modifier.

The C syntax for the use of execution-class modifier in a function declaration to include execution-class names in the function declaration may optionally appear only once for each function declaration.

5.4.1 Detailed changes to ISO/IEC 9899:1999

This clause details the modifications to ISO/IEC 9899:1999 needed to incorporate the functionality of execution-class specifiers overviewed in Clauses 5.4 of this Technical Report. The changes listed in this clause are limited to syntax and semantics; examples, (forward) references and other descriptive information are omitted. The modifications are ordered according to the clauses of ISO/IEC 9899:1999 to which they refer. If a clause of ISO/IEC 9899:1999 is not mentioned, no changes to that clause are needed. New clauses are indicated with **(NEW CLAUSE)**; resulting changes in the existing numbering are not indicated. Clauses number *mm.nn*a of new clause indicates that this clause follows immediately clause *mm.nn* at the same level.

Clause 6.7.4.1 – Function Modifier (NEW CLAUSE)

Function Specifiers This will add execution class specifiers as a function specifier.

Syntax

1 *function-Modifier:*

execution-class

Constraints

2 Function specifiers shall be used only in the declaration of an identifier for a function.

Semantics

3 A function declared with an execution-class function modifier will execute on the named execution unit. The function modifier may optionally appear only once for each function declaration.

4 Any function with internal linkage can have a processor modifier. For a function with external linkage, the processor modifier remains as the original function was declared.

EXAMPLE

5 The declaration of an execution class specifier function modifier with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with **extern** creates an external definition. The following example shows an execution class function modifier.

```
function_modifier double fahr(double t)
{
  return (9.0 * t) / 5.0 + 32.0;
}
```

Additions to Annex B

B.2 Execution address space support

B.2.1 Execution address space modifiers

Execution space modifiers may have implementation defined optional arguments. The optional arguments are available to tie the execution unit-specific information to a new name defined by the _Processor definition statement.

__Processor ***newnam*****{[*****n*****]} = *****processor_id*** **{,*****optional arguments*****};**