The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

5.2.3 Signals and interrupts

Commentary 270

signal

Rationale

Signals are difficult to specify in a system-independent way. The C89 Committee concluded that about the only thing a strictly conforming program can do in a signal handler is to assign a value to a volatile static variable which can be written uninterruptedly and promptly return.

. . .

A second signal for the same handler could occur before the first is processed, and the Standard makes no guarantees as to what happens to the second signal.

WG14/N748

A pole exception is the same as a divide-by-zero exception: a finite non-zero floating-point number divided by a zero floating-point number.

Currently, various standards define the following exceptions for the indicated sample floating-point operations. For LIA–2, there are other operations that produce the same exceptions.

LIA	<----------- Standard		----------------->	IEEE
Exception	LIA-1	LIA-2	IEEE-754/IEC-559	Exception
undefined	0.0 / 0.0	sqrt(-1.0)	0.0 / 0.0	invalid
	1.0 / 0.0	log(-1.0)	infinity / infinity	
			infinity - infinity	
			0.0 * infinity	
			sqrt(-1.0)	
pole	(not yet)	log(0.0)	1.0 / 0.0	division by zero
floating_ overflow	max * max	exp(max)	max * max	overflow
	max / min		max / min	
	max + max		max + max	
underflow	min * min	exp(-max)	min * min	underflow
	min / max		min / max	

In the above table, 1.0/0.0 is a shorthand notation for any non-zero finite floating-point number divided by a zero floating-point number; max is the maximum floating-point number (`FLT_MAX`, `DBL_MAX`, `LDBL_MAX`); min is the minimum floating-point number (`FLT_MIN`, `DBL_MIN`, `LDBL_MIN`); `log()` and `exp()` are mathematical library routines.

We believe that LIA–1 should be revised to match LIA-2, IEC-559 and IEEE-754 in that 1.0/0.0 should be a pole exception and 0.0/0.0 should be an undefined exception.

C++

The C++ Standard specifies, Clause 15 Exception handling, a much richer set of functionality for dealing with exceptional behaviors. While it does not go into the details contained in this C subclause, they are likely, of necessity, to be followed by a C++ implementation.

Other Languages

Some languages (e.g., Ada, Java, and PL/1) define statements that can be used to control how exceptions and signals are to be handled. After over 30 years floating point exception handling has finally been specified in the Fortran Standard.[2] A few languages include functionality for handling signals and interrupts, but most ignore these issues.

Common Implementations

Implementations are completely at the mercy of what signals are supported by the host environment and what interrupts are generated by the processor. Gould (Encore) PowerNode treated both floating-point and integer overflow as being the same.

Coding Guidelines

This subclause lists those minimum characteristics of a program image needed to support signals and interrupts. Such support by the implementations is only half of the story. A program that makes use of signals has to organize its behavior appropriately. Techniques for writing programs to handle signals, or even ensuring that they are thread-safe are outside the scope of these coding guidelines.

---

271 Functions shall be implemented such that they may be interrupted at any time by a signal, or may be called by a signal handler, or both, with no alteration to earlier, but still active, invocations' control flow (after the interruption), function return values, or objects with automatic storage duration.

Commentary

This is a requirement on the implementation. An implementation may provide a mechanism for the developer to switch off interrupts within time-critical functions. Although such usage is an extension to the standard, it cannot be detected in a strictly conforming program.

How could an implementation's conformance to this requirement be measured? A program running under an implementation that supports some form of external interrupt, for instance `SIGINT`, might be executed a large number of times, the signal handler recording where the program was interrupted (this would require functionality not defined in the standard). Given sufficient measurements, a statistical argument could be used to show that an implementation did not support this requirement. A nonprogrammatic approach would be to verify the requirement by understanding how the generated machine code interacted with the host processor and the characteristics of that processor.

This wording is not as restrictive on the implementation as it first looks. The only signal that an implementation is required to support is the one caused by a call to the `raise` function. Requiring that any developer-written functions be callable from a signal handler restricts the calling conventions that may be used in such a handler to be compatible with the general conventions used by an implementation. This simplifies the implementation, but places a burden on time-critical applications where the calling overhead may be excessive.

C++

This implementation requirement is not specified in the C++ Standard (1.9p9).

Other Languages

Most languages don't explicitly say anything about the interruptibility of a function.

Common Implementations

Few if any host processors allow execution of instructions to be interrupted. The boundary at the completion of one instruction and starting another is where interrupts are usually responded to. In the case of pipelined processors, there are two commonly seen behaviors. Some processors wait until the instructions currently in the pipeline have completed execution, while others flush the instructions currently in the pipeline. An example of an instruction that causes an interrupt to be raised after it has only partially completed is one that accesses storage, if the access causes a page fault (causing the instruction to be suspended while the accessed page is swapped into storage). Another case is performing an access to storage using a misaligned address, or an invalid address. In these cases the instruction may never successfully complete.

External, nonprocessor-based interrupts are usually only processed once execution of the current instruction is complete. Some processors have instructions that can take a relatively long time to execute, for instance, instructions that copy large numbers of bytes between two blocks of memory. Depending on the design requirements on interrupt latency, some processors allow these instructions to be interrupted, while others do not.

Some implementations[4] require that functions called by a signal handler preserve information about the state of the execution environment, such as register contents. Developers are required to specify (often by using a keyword in the declaration, such as **`interrupt`**) which functions must save (and restore on return) this information.

---

object storage outside function image

All such objects shall be maintained outside the *function image* (the instructions that compose the executable representation of a function) on a per-invocation basis. 272

Commentary

This is a requirement on the implementation (although the as-if rule might be invoked). The model being described is effectively a stack-based approach to the calling of functions and the handling of storage for objects they define (the actual storage allocation could use a real stack or simulate one using allocated storage).

Storing objects in the function image, or simply having a preallocated area of storage for them, would prevent a function from being called recursively (having more than one call to a function in the process of being executed at the same time is a recursive invocation, however the invocation occurred). An implementation is required to support recursive function calls. This requirement prevents implementations using a technique that was once commonly used (primarily by implementations of other languages), but can have different execution time semantics when recursive calls are made.

function call recursive

C++

The C++ Standard does not contain this requirement.

Other Languages

function call recursive

Most languages require support for recursive function calls, implying this requirement.

Common Implementations

Modern processors try to separate code (function image) and data (object definitions). Accesses to the two have different characteristics, which affects the design of caches for them (often implemented as two separate cache areas on the chip). Independently of processor support, the host environment (operating system) may mark certain areas of storage as having execute-only permission. Attempts to read or write to such storage, from an executing program, often leads to a signal being raised.

Applications targeted at a freestanding environment rarely involve recursive function calls. Storage may also be at a premium and hardware stack support limited (the Intel 8051[1] is limited to a 128-byte stack). Some hosts allocate fixed areas, in static storage, for objects local to functions. A call tree, built at link-time, can be used to work out which storage areas can be shared by overlaying those objects whose lifetimes do not overlap, reducing the fixed execution time memory overhead associated with such a design.

Many processors have span-dependent load and store instructions. That is, a short-form (measured in number of bytes) that can only load (or store) from/to storage locations whose address has a small offset relative to a base address, while a long-form supports larger offsets. When storage usage needs to be minimized, it may be possible to use a short-form instruction to access storage locations in the function image. The usual technique used is to reserve storage for objects after an unconditional branch instruction, which is accessed by the instructions close (within the range supported by the short-form instruction) to those locations.[3]

Coding Guidelines

While implementations might be required to allocate objects outside of a function image, developers have been known to write code to store values in a program image. In those few cases where values are stored in this way, the developers involved are very aware of what they are doing. A guideline recommendation serves no purpose.

Example

The following is one possible method that might be used to store data in a program image.

```
#include <stdio.h>

extern int always_zero = 0;
static int *code_ptr;

void f(void)
{
/*
 * No static object declarations in this function ;-)
 */
if (always_zero == 1) /* create some dead code */
   {
   /*
    * Pad out with enough code to create storage for an int.
    * A smart optimizer is the last thing we need here.
    */
   always_zero++;
   always_zero++;
   }

(*code_ptr)++;
printf("This function has been called %d times.\n", *code_ptr);
}

void init(void)
{
/*
 * The value 16 is the offset of the dead code from the start of the
 * function.  Change to suit your local instruction sizes (this works
 * for gcc on an Intel x86).  We also need to make sure that the
 * pointer to int is correctly aligned.  A reliable guess is that
 * the alignment is a multiple of the object size.
 */
code_ptr=(int *)((((int)(char *)f) + 16) & ~(sizeof(int)-1));
*code_ptr=0;
}

int main(void)
{
init();
for (int index=0; index < 10; index++)
   f();
}
```

References

1. Intel. *MCS 51 Microcontroller Family User's Manual*. Intel, Inc, 272383-002 edition, Feb. 1994.
2. ISO. *ISO TR 15580:1998 Information technology —Programming languages —Fortran —Floating-point exception handling*. ISO, 1998.
3. E. L. Robertson. Code generation and storage allocation for machines with span-dependent instructions. *ACM Transactions on Programming Languages and Systems*, 1(1):71–83, 1979.
4. Texas Instruments. *TMS370 and TMS370C8 8-Bit Microcontroller Family Optimizing C Compiler Users' Guide*. Texas Instruments, spnu022c edition, Apr. 1996.