A `pre_main()` function that runs before static initialization that precedes `main()`

Document #:	D9999
Date:	February 3, 2015
Project:	Programming Language C++ Evolution Working Group

Reply-to:	Sébastian Davalle <sebastian.davalle@tallertechnologies.com>
	Daniel Gutson <daniel.gutson@tallertechnologies.com>
	Christopher Kormanyos <e_float@yahoo.com>

1 Motivation

Overview

Many systems require user-specific initialization. In particular, deeply embedded environments may have user-specific initialization that should be executed as soon as possible following power-on-reset of the microcontroller. Typical examples thereof include initialization of I/O ports, watchdog timers, instruction and data cache systems, interrupt systems, clock systems, etc. For these environments, postponing user-specific initialization until `main()` may detract from quality of design. Another important use is setting up tools such as logging or allocator mechanisms—notoriously meticulous activities that could potentially be simplified with a `pre_main()` initialization.

2 Proposed Solution

This document proposes changes in the standard text reflecting the specification of a `pre_main()` function that is intended to be called prior to static initialization that precedes `main()`.

Modification to the standard text

1. **[basic.start.main] Add a new paragraph §3.6.1p6**

   The `pre_main()` function shall be called prior to *static initialization* (§3.6.2) that precedes `main`. The linkage (3.5) of `pre_main()` is implementation-defined. The `pre_main()` function is parameter-free. The return value of `pre_main()` is `void`. The `pre_main()` function lacks

side-effects. The mechanism ensuring that `pre_main()` lacks side-effects is implementation-defined.

3 Existing workarounds

We will now investigate existing workarounds for providing a call mechanism for `pre_main()` or a similar call mechanism.

Sample startup code

Embedded systems developers and compiler implementers often write startup code. In this case, it is straight forward to support a `pre_main()` function. For example, we will now look at sample startup code [3, 4] showing how an implementation could potentially provide a call mechanism for `pre_main()`.

```
void __my_startup()
{
  // Load the sreg register.
  asm volatile("eor r1, r1");
  asm volatile("out 0x3F, r1");

  // Setup the stack pointer.
  asm volatile("ldi r28, lo8(__initial_stack_pointer)");
  asm volatile("ldi r29, hi8(__initial_stack_pointer)");
  asm volatile("out 0x3E, r29");
  asm volatile("out 0x3D, r28");

  // A potential call mechanism for pre_main.
  pre_main();

  // Initialize statics from ROM to RAM.
  // Initialize default-initialized static RAM.
  crt::init_ram();

  // Call all ctor initializations.
  crt::init_ctors();

  // Call main (and never return).
  asm volatile("call main");

  // Catch an unexpected return from main.
  for(;;)
  {
    // Replace with a loud error if desired.
```

```
    mcal::wdg::secure::trigger();
  }
}
```

This example has been taken from the low-level initialization sequence of a popular 8–bit microcontroller. The code has been compiled and tested with GCC 4.8.1 [1]. After setting a CPU register, the stack pointer is initialized. Immediately following stack setup, `pre_main()` is called. Note that `pre_main()` is called prior to static initialization.

Commercially available microcontroller compilers

Some commercially available microcontroller compilers provide a custom hook (in the sense of `pre_main()`) that is called before static initialization that precedes `main()`. The IAR Systems C/C++ compiler and debugger toolchain [2], for instance, uses an implementation-specific function called `__low_level_init()` for this purpose. The user is responsible for supplying the content (if any) of `__low_level_init()`.

4 Future Work

The motivation and justification for a potential `pre_main()` is analogous in C and C++. Therefore, specifying `pre_main()` could potentially be addressed in WG14 as well as WG21.

Along these lines, do we need two versions of `pre_main()`? In particular,

```
void     ::pre_main(void); // Intended for C/C++
void std::pre_main();      // Intended for C++
```

What is the proper name of a potential `pre_main()`? Is `pre_init()` a better name because it more clearly reflects when the function is called?

Despite the proposal that `pre_main()` lacks side-effects, it could be beneficial to allow `pre_main()` to initialize certain *clearly identifiable* non-local variables having static storage duration. Embedded systems tool chains for C/C++ typically provide special linker sections with implementation-specific names such as `.noinit`, `.noclear`, etc. These are meant to store non-local variables having static storage duration that are not intended to undergo static initialization. Attributes such as `[[noclear]]` or `[[noinit]]` could be used to clearly identify these.

Consider, for example, the `reset_reason` in the following code.

```
typedef enum enum_reset_reason
{
  power_on_reset,
  watchdog_reset,
  software_reset
}
reset_reason_type;
```

```
[[noclear]] reset_reason_type reset_reason;
```

Here, `reset_reason` is intended to be initialized by `pre_main()` in the application, not via conventional static initialization.

5 Discussion

TBD: Summarize the discussion regarding when `pre_main()` should be called and why.

TBD: Summarize the discussion regarding the dangers of offering an open user-interface that precedes `main()`. Will users run into inordinate amounts of trouble with this proposed interface?

TBD: Specify what can be done and not inside `pre-main()`

TBD: Distinguish the usage in terms of freestanding/hosted implementation.

TBD: Define what should be done in a failure condition.

TBD: Exception handling (nonexcept attribute)

TBD: Calls to `std::` functions is allowed or not?

TBD: calls to `atexit()`, `exit()`, `at_quick_exit()`, `quick_exit()` should be undefined.

TBD: call to `terminate()` should be implementation specific.

6 Acknowledgments

TBD: Acknowledge the participants.

7 References

[1] Free Software Foundation: *GNU Compiler Collection*
http://gcc.gnu.org (2015)

[2] IAR Systems, *IAR Embedded Workbench® C/C++ compiler and debugger toolchain*,
http://www.iar.com/Products/IAR-Embedded-Workbench (2015)

[3] C. M. Kormanyos, *Real-Time C++*, Springer Verlag, Heidelberg, 2013

[4] C. M. Kormanyos, *real-time-cpp : Companion code for Real-Time C++*, real-time-cpp, 2015