Generic access_type descriptor

for the Embedded C Technical Report

by
Jan Kristoffersen
Walter Banks

Purpose:

This document proposes a consistent and complete specification syntax for defining I/O registers and their access methods in C.

Background:

Current work has shown that there are three basic requirements which must not be compromised by any standardized solution for portable I/O register access:

- The symbolic I/O register name used in the I/O driver code must refer to a **complete definition of the access method**.
- The standardized solution must be able to **encapsulate** all knowledge about the underlying processor, platform, and bus system.
- It should provide a **no-overhead solution** (for simple access methods).

In order to fulfill the first two requirements in a consistent way, it should be possible *to refer to a complete access_type specification as a single entity*. This is necessary, for instance, to pass *access_type* parameters between functions.

This can been achieved in several different ways. Prior art has used a number of (intrinsic) memory type qualifiers or special keywords, which have varied from compiler to compiler and from platform to platform.

However, type qualifiers have always tended to be an inadequate description method when more complex access methods are needed. For instance, it must be possible to encapsulate all access method variation possible in the target platform. These differences include the widths of I/O registers, and the qualities of the I/O chip bus and processor bus: register interleave values, I/O register endian specifications, and so on. Similarly, type qualifiers are usually inadequate when more complex addressing methods are used (base pointer addressing, pseudo-bus addressing, addressing via user device drivers, and others).

This paper proposes a generic syntax for defining the *access_type* for an I/O register. The syntax is a new approach and a super-set solution, intended to replace prior art based on type qualifiers and/or the @ operator.

Syntax specification

Access_type specification:

typedef ACCESS_METHOD_NAME < parameter list > SYMBOLIC_PORT_NAME;

parameter list:
access method independent parameter list , access method specific parameter list

access method independent parameter list:
type for I/O register value (size of I/O register) ,
access limitation type ,
I/O register chip bus type (size and endian of I/O chip bus)

type for I/O register value (size of I/O register):
uint8_t
uint16_t
uint32_t
uint64_t
bool
(+ optionally any basic type native to the implementation)

access limitation type: // for compile time diagnostic
ro_t *//read_only*
wo_t *//write_only*
rw_t *//read_write*
rmw_t *//read_modify_write*

I/O register chip bus type:
chip8 *// register width = chip bus width = 8 bit*
chip8l *// register width > chip bus width, MSB on low address*
chip8h *// register width > chip bus width, MSB on high address*
chip16 *// register width = chip bus width = 16 bit*
chip16l *// register width > chip bus width, MSB on low address*
chip16h *// register width > chip bus width, MSB on high address*
chip32 *// register width = chip bus width = 32 bit*
chip32l *// register width > chip bus width, MSB on low address*
chip32h *// register width > chip bus width, MSB on high address*
chip64 *// register width = chip bus width = 64 bit*
(+ optionally any bus width native to the implementation)

access method specific parameter list:
// Depends on the given access method. Examples are given later.
// Three typical parameters are:
Primary address constant ,
Processor bus width type,
Address mask constant

Processor bus width type:
bw8 *// 8 bit bus*
bw16 *// 16 bit bus*
bw32 *// 32 bit bus*
bw64 *// 64 bit bus*
(I.e. any bus widths native to the implementation)

Angled brackets are employed as delimiters for the access type parameter list, to distinguish it from a function parameter list (using "()") and from a structure or enumerated list (using "{}").

An implementation must define at least one access method for each processor addressing range.
For instance, for the 80x86 CPU family, an implementation must define at least two access_methods, one for the memory-mapped range, and one for the I/O-mapped range.
If several different access methods are supported for a given address range, then an access type must exist for each access method.

The *ACCESS_METHOD_NAME* is an identifier for the parameter set enclosed in angled brackets. It is an implementation-defined keyword which tells the compiler how to interpret the parameter set. A compiler will typically support a number of different *access_type* descriptors.

Examples of access_type descriptors

Below are some examples of *access_type* parameter combinations for different (typical) access methods:

Direct addressing:

```
typedef MM_DIRECT <
       type for I/O register value (size of I/O register) ,
       access limitation type ,
       I/O register chip bus type (size and endian of I/O chip bus),
       primary address constant,
       processor bus width type
       > PORT_NAME;
```

The I/O register at the primary address is addressed directly. If the bit width of the I/O register is larger than the I/O chip bus width, then the access operation is built from multiple consecutive addressing operations.

Based addressing:

```
typedef MM_BASED <
       type for I/O register value (size of I/O register) ,
       access limitation type ,
       I/O register chip bus type (size and endian of I/O chip bus),
       primary address constant,
       processor bus width type,
       base variable
       > PORT_NAME;
```

The I/O register at the *primary_address* + value of *base_variable* is addressed directly. If the bit width of the I/O register is larger than the I/O chip bus width,

then the access operation is built from multiple consecutive addressing operations.

Indexed-bus addressing:

```
typedef MM_INDEXED <
       type for I/O register value (size of I/O register) ,
       access limitation type ,
       I/O register chip bus type (size and endian of I/O chip bus),
       primary address constant,
       processor bus width type,
       secondary address parameter
       > PORT_NAME;
```

The I/O register on an indexed bus (also called a pseudo-bus) is addressed in the following way. The primary address is written to the register given by the secondary address parameter (= initiate indexed bus address). The access operation itself is then done on the location (secondary address parameter+1 = data at indexed bus).
This method is a common way to save addressing bandwidth. The method also makes it particularly easy to connect chips using a multiplexed address/data bus interface to a processor system having a non-multiplexed interface.

Device driver addressing:

```
typedef MM_DEVICE_DRIVER <
       type for I/O register value (size of I/O register) ,
       access limitation type ,
       I/O register chip bus type (size and endian of I/O chip bus),
       primary address constant,
       processor bus width type,
       name of driver function for register write,
       name of driver function for register read
       > PORT_NAME;
```

The I/O register is addressed by invoking (user-defined) driver functions. If the bit width of the I/O register is larger than the I/O chip bus width, then the access operation is built from multiple consecutive addressing operations.
(Alternatively, the I/O register chip bus type, processor bus width type and the primary address could be transferred to the driver functions.)

Direct bit addressing:

```
typedef MM_BIT_DIRECT <
       type for I/O register value (size of I/O register) ,
       access limitation type ,
       I/O register chip bus type (size and endian of I/O chip bus),
       primary address constant,
       processor bus width type,
       bit location in register constant,
       > PORT_NAME;
```

The I/O register at the primary address is addressed directly.

Examples:

```
typedef MM_DIRECT <uint8_t,rw_t,chip8,0x3000,bw8> MYPORT;
uint8_t a = iord(MYPORT,0xAA); // Read single register
```

MYPORT is an 8-bit read-write register, located in a chip with an 8-bit data bus, connected to a (memory-mapped) 8-bit processor bus at address 0x3000.

```
typedef MM_DIRECT <uint16_t,wo_t,chip8l,0x200,bw16> PORTA;
iowr(MYPORT,0xAA); // Write single register
```

PORTA is a 16-bit write-only register, located in a chip with an 8-bit data bus (with MSB register part located at the lowest address), where the chip is connected to a (memory-mapped) 16-bit processor bus at address 0x200.

Use of user-defined device drivers:

```
// Memory buffer addressed via user-defined access drivers
typedef MM_DEVICE_DRIVER
     <uint8_t,rmw_t,chip8,0xA,my_wr_drv,my_rd_drv> DRVREG;

// User-defined read driver to be invoked by compiler
inline uint8_t void my_rd_drv( int index )
   {
   // some driver code
   }

// User-defined wr driver to be invoked by compiler
inline void my_wr_drv( int index , uint8_t dat)
   {
   // some driver code
   }

// user code
int i;
i = iord(DRVREG);           // = call of my_rd_drv(0xA);
for (i = 0; i < 0xA0; i++)
    iowrbuf(DEVREG,i,0x0);  // = call of my_wr_drv(i+0xA,0)
```

Parsing

The access type descriptors are parsed at compile time.

If the symbolic port name is used directly in iord(..)/iowr(..)/etc. functions, the code can be completely optimized at compile time: all information for doing this is available to the compiler at that stage. Based on the combined parameter set (the types), the compiler will typically select among several internal intrinsic inline access functions to generate the appropriate code for the access operation. No memory instantiation of an *access_type* object is needed. This will fulfill the third of the primary requirements on page 1 (no-overhead solution).

Example:

```
typedef MM_DIRECT<uint16_t,rmw,chip8l,0x3456,bw16> MY_PORT1;
uint16_t d;
//...
d = iord(MY_PORT1);   // no-overhead in-line code
iowr(MY_PORT1, 0x456);
```

If the symbolic port name is referenced via a pointer, then an *access_type* object must be instantiated in memory; (slower) generic functions are invoked by the iord(..)/iowr(..)/etc. functions. In this case, the *access_type* parameter is mostly evaluated at runtime. (This approach is similar to the one used for *extern inline* functions in C)

Example:

```
typedef MM_DIRECT<uint16_t,rmw_t,chip8l,0x3456,bw16> MY_PORT1;
typedef MM_DIRECT<uint16_t,ro_t,chip16l,0x7890,bw16> MY_PORT2;

uint16_t foo(MM_DIRECT * iop)
    {
    return iord(iop); // invoke some generic iord function
    }

uint16_t a;
a = foo(MY_PORT1);
a+=foo(MY_PORT2);
```

Embedded systems extended memory support

Many embedded systems include memory that can only be accessed with some form of device driver. These include memories accessed by serial data busses (I^2C, SPI), and on-board non-volatile memory. Device driver memory support is used in applications where the details of the access method can be separated from the details of the application.
In contrast to memory-mapped I/O, the extended memory layout and its use should be administrated by the compiler/linker.
Language support for embedded systems need to address the following issues:

1. Memory with user-defined device drivers. User-defined device drivers are required for reading and writing user-defined memory.
   - Memory read functions take as an argument an address in the user-defined memory space, and return data of a user-defined size.
   - Memory write takes two arguments an address in the user-defined memory space and data with a user-specified size.
   - Applications require support for multiple user-defined address spaces.
   - User-defined memory areas may not be contiguous. Most of the applications have gaps in the addressing within user-defined memory areas.

2. The compiler is responsible for:
   - Allocating variables, according to the needs of the application, in “normal” address space and in space accessed by the user-defined memory device drivers.
   - Making calls to device drivers, when accessing variables supported by user-defined device drivers.
   - Automating the process of casting and accessing the data, between calls to access data and the application.

3. Application variables in user-defined memory areas :
   - Need to support all of the available data types. For example, declarations for fundamental data types, arrays, structures.
   - Users need to direct the compiler to use a specific memory area.
   - The compiler needs to be free to use user-defined memory area as a generic, general-purpose memory area, for the purposes of a variable spill area.

The following declaration shows all the information that is needed to declare memory for use with user-defined device drivers.

```
typemod USER_MEMORY <
   access limitation type ,
   device driver for data read,
   device driver for write,
   primary address constant,   // Base address of memory in
                               // the drivers address space
   address range               // Size of memory handled by
                               // the device drivers
  [optional additional address range definitions]
 > memory_name;
```

The **typemod** definition is a method of encapsulating the memory declaration. **typemod** ties variable declarations to device drivers, and provides the compiler a means of using data that the user provides to manage variables that are required by an application. User-defined memory may be global in nature, or local to one program segment.

```
typemod USER_MEMORY <rmw_t,
                     ddram_r,
                     ddram_w,
                     0x90,0x30
                     0xD0,0x30
                     > ddram

/* A typemod definition always specifies a linear memory (fragment(s))*/

/* function prototypes to read and write user-defined memory.
    It is the responsibility of the device driver to transfer
    the number of bytes requested. An optimizing compiler
    can pass structures or data, and the driver will
    optimize the transfer */


    void ddram_w( int location, char *src, int size);
    void ddram_r( int location, char *desc, int size);

char a;   /* normal memory declarations */
int b;
long c;

// Modifier puts variable in user-named address space

char ddram wa;
int ddram wb;
long ddram wc;
char ddram ar[10];
unsigned int ddram wc;

wa = 0x33; // ddram_w called (must be stored as an int)
a = wa;    // ddram_r called once, MSB turncated
b = wb;    // ddram_r called once
c = wc;    // ddram_r called more than once (implementation defined)
```

Comments on syntax notation

The syntax notation is inspired by the syntax used for template-based implementations of *iohw* in C++. However, this does not in any way imply that C++ features like templates and overloading should be added to the language. It is merely a new syntax for specifying the complex entity of an *access_type* (complex in the sense that it must consist of multiple entities).

The advantages with the proposed notation are: that it can be made reasonable consistent across processor and bus architectures, and (most importantly) it will be both fairly easy to comprehend and to use for the average embedded programmer. (In contrast to this are pure macro-based implementations, which tend to become rather complex to understand, create, and maintain for the user.)

The header file which defines the hardware will look simple (typically, like a list, with one register definition per text line). This makes it easy for a user to adapt an existing *access_type* definition to new hardware. Maintenance becomes much simpler.