C++ Interrupts in avr-gcc

This document is in support of the patch to add a numeric parameter to the gcc signal function.

While most embedded projects use C as the language of choice, there are some large projects where the extra control provided by using class structures would be a benefit. While a C++ program will usually require more program memory, the difference is not that large. The problem is usually not the additional flash space required but how to handle interrupts.

There are two separate but related problems:

1. Name mangling in C++.
2. Accessing private class data.

Name Mangling

Avr-gcc defines an interrupt vector as a name that the linker will use to overide the weak reset addresses in the AVR interrupt vector table. For example, the USART0 receiver interrupt on an ATmega128 is defined as:

```
#define USART0_RX_vect        _VECTOR(18)
```

The _VECTOR(18) macro is expanded to:

```
#define USART0_RX_vect      __vector_18
```

and so the ISR macro used to define the corresponding interrupt handler in code expands to:

```
void __vector_18(void) __attribute__ ((signal, __INTR_ATTRS));
```

and this wll compile into the assembler output as the two lines:

```
.global  __vector_18
__vector_18:
```

The linker picks up the name as matching the corresponding interrupt vector table entry and makes the replacement into the vector table. Thus an interrupt with this number will arrive at the corresponding interrupt handler.

However in C++ the name is mangled early in the compile process and prevents the linking from occurring. This patch allows an interrupt handler name to be represented by a number, and while the name will be mangled as usual, the number survives for later linking. The patch provides for an optional numeric argument to the **signal** function. An interrupt function prototype using this system for the same USART0 receiver interrupt looks like:

```
void IntName(void) __attribute__ ((signal(18), __INTR_ATTRS));
```

The numeric signal argument is translated by this patch into the same two assembler lines as above. The given name is still processed according to the language rules. The name is thus

independent of the vector number, but the number is attached to the name. Note that for C++, by the time the signal argument is being processed the given name is mangled.

Once implemented the patch can be used, but to be versatile it will require an additional definition for each interrupt in each processor. The current proposal is to add the new definition along with those existing. For example, the USART interrupt above for the '128 in file iom128.h will now have two lines.

```
#define USART0_RX_vect          _VECTOR(18)
#define USART0_RX_vect_num      18
```

The corresponding new interrupt macro for C++ interrupt declarations is defined in **Interrupts.h** as:

```
#define ISRN(name, number) void name(void) __attribute__ ((signal(number),
__INTR_ATTRS));
```

Here is an example to show how this macro is used in defining an interrupt class. We will discuss how this can be used in the next section.

```
class CUsart0Interrupt
{
public:
   CUsart0Interrupt();
   ~CUsart0Interrupt();
private:
   ISRN(RxInterrupt, USART0_RX_vect_num);
};
```

Accessing Private Class Data

While we can now define interrupt handlers within a device management class and declare them private, there is still no way they can access the class's private data because the class itself has no access to its own data pointer, the so-called *this* pointer in C++. The *this* pointer is defined where the class is declared, and in avr-gcc is loaded into one of the 16-bit AVR pointers prior to calling a class method. An interrupt cannot know this pointer and has no way of finding it, and so we need some way to enter the class normally (ie with a this pointer) from its various interrupt handlers.

In C++ you can declare one or more classes as *friends* of another class, and this gives them access to both private and public methods and variables in that class. We can therefore define a class that manages a device (Usart, Timer, AtoD converter, etc) and define another class we will call its friend that contains all that devices' interrupt handlers. Access to the private data from the friend class is important for speed, as just calling a function in the device class from the interrupt class will have avr-gcc stack almost all its registers. The mechanism described above vectors the interrupt into the interrupt class, and accessing the device class (that is managing the peripheral and its data) gives us access to its *this* pointer.

Let us define a class that manages a particular AVR peripheral, for example the Timer0. We will call the device class **CTimer0** and its interrupt handler class **CTimer0Interrupt**. We will keep it simple and define the class as having an overflow interrupt that sets a flag, a test method that will return the flag's state, and a reset method that will clear the flag for subsequent use. First the interrupt class. The example **CUsart0Interrupt** class defined above

included a constructor and destructor. These could be used to setup the hardware, flags, etc associated with the interrupt, but because an interrupt class may contain several handlers, I prefer to leave all initialisation in the device class. There is no requirement for a constructor or destructor.

```
//---------------------------------------------------------------------
// The class definition for all interrupts associated with Timer0.
// Any and all of the Timer0 interrupts are defined here.
//
class CTimer0Interrupt
{
private:
   ISRN(OverflowInterrupt, TIMER0_OVF_vect_num);    // overflow interrupt
//   ISRN(CompareInterrupt, TIMER0_COMP_vect_num);  // (another example)
};
```

The device class defines the interrupt handler class as a friend. Note how the interrupt handler itself is actually private class data, and the **mFlag** member variable is qualified as *volatile* because it will be accessed by the interrupt handler.

```
//---------------------------------------------------------------------
// The class definition for all functions associated with Timer0.
// The CTimer0Interrupt class is declared a friend so it has direct
// access to the mFlag private variable.
//
class CTimer0
{
   friend class CTimer0Interrupt;
public:
   CTimer0();                  // constructor
   ~CTimer0();                 // destructor
   int GetFlag(void);          // test flag state
   void ResetFlag(void);       // clear flag state

private:
   volatile int mFlag;         // interrupt flag
   CTimer0Interrupt Timer0Int; // device interrupt handler
};
```

Both these classes are normally contained in a header file. The associated code file (cpp) is given below.

```
#include "Timer0.h"

CTimer0 Timer0;   // the CTimer0 instance

//---------------------------------------------------------------------
// The constructor provides all the required initialisation for
// this device.
//
CTimer0::CTimer0()
{
   TCNT0 = TIMER0_TIMEOUT;
   TCCR0 = 0x04;
   TIMSK |= SB(TOIE0);        // enable overflow interrupts
   mFlag = 0;
}

//---------------------------------------------------------------------
// Unlikely to be used in an embedded app.
//
CTimer0::~CTimer0()
{
```

```
   TIMSK &= ~SB(TOIE0);       // disable Timer0 timeout interrupt
}

//---------------------------------------------------------------------
// Probe to check if an interrupt has occurred. Should really use
// an atomic access that saves the processor interrupt state.
//
int CTimer0::GetFlag(void)
{
   int flag;

   cli();
   flag = mFlag;
   sei();
   return flag;
}

//---------------------------------------------------------------------
// Reset the interrupt flag. Should also be interrupt safe.
//
void CTimer0::ResetFlag(void)
{
   cli();
   mFlag = 0;
   sei();
}

//---------------------------------------------------------------------
// The overflow interrupt handler. Note it is only a friend of the
// device class, but has direct access to class data members. Note
// that it must know the name of the CTimer0 class instance.
//
void CTimer0Interrupt::OverflowInterrupt(void)
{
   TCNT0 = TIMER0_TIMEOUT;    // restart the timeout
   Timer0.mFlag = 1;          // set the mFlag in CTimer0
}
```

Finally the main program. This is just a skeleton but shows how the device class state and reset methods are used.

```
#include "Main.h"
#include "Timer0.h"

volatile int Count;

int main(void)
{
   Count = 0;
   sei();
   Timer0.ResetFlag();

   while (TRUE)
   {
      if (Timer0.GetFlag() != 0)
      {
         Timer0.ResetFlag();
         Count++;
      }
   }
}
```

23:07 11-Apr-2007, Ron Kreymborg