A Reusable Duff Device

Loop unrolling made easy

RALF HOLLY

In embedded programming, especially when developing device driver-level code, performance is often at a premium. One way to improve execution time is to reduce the overhead of looping through the use of an age-old technique known as "loop unrolling."

Tom Duff invented a special kind of loop-unrolling mechanism, known as "Duff's Device," that makes loop unrolling much easier. In this article, I examine the idea behind loop unrolling and explain how to generalize loop unrolling such that it can be put in a reusable library.

Classic Loop Unrolling

When implementing device drivers, you often have to access device registers a certain number of times from within a loop. Imagine a network I/O driver that sends characters to a port:

```
#define HAL_IO_PORT
                *(volatile char*)0xFFFF8000
for (i = 0; i < len; ++i) {
    HAL_IO_PORT = *pSource++;
}
```

Ralf is principal of PERA Software Solutions and can be contacted at rholly@pera-software.com.

For each pass through the loop (that is, for each character copied/sent to the port), three things happen:

- There is a jump to the beginning of the loop.
- Next *i*, the loop counter, is incremented.
- *i* is compared against *len*.

If the output operation itself consumes very little time (as it is normally the case when accessing a device register), these three steps add significant overhead.

Instead of copying the characters in a loop, you can transmit the characters "inline" and thereby save the jump, increment, and comparison:

```
HAL_IO_PORT = *pSource++;
HAL_IO_PORT = *pSource++;
HAL_IO_PORT = *pSource++;
HAL_IO_PORT = *pSource++;
HAL_IO_PORT = *pSource++;
HAL_IO_PORT = *pSource++;
 ...
```

However, this approach suffers from two weaknesses. First, it is static, meaning you have to know exactly how many characters you want to transfer; second, it is wasteful in terms of code memory, because the same line of code is repeated *len* times.

As a result, developers usually implement loop unrolling like this:

```
int n = len / 8;
for (i = 0; i < n; ++i) {
    HAL_IO_PORT = *pSource++;
    HAL_IO_PORT = *pSource++;
    HAL_IO_PORT = *pSource++;
    HAL_IO_PORT = *pSource++;
    HAL_IO_PORT = *pSource++;
    HAL_IO_PORT = *pSource++;
    HAL_IO_PORT = *pSource++;
    HAL_IO_PORT = *pSource++;
}
```

This approach processes eight operations in a row, which reduces the looping overhead to one-eighth. To handle cases were *len* is not evenly divisible by eight, a postprocessing loop is needed to execute the remaining copy operations:

> "Tom Duff invented a special kind of loop-unrolling mechanism known as 'Duff's Device'"

```
n = len % 8;
for (i = 0; i < n; ++i) {
    HAL_IO_PORT = *pSource++;
}
```

Even though this scheme works well, it is quite "wordy" and definitely not easy to maintain. Further, it cannot be generalized and put in a library.

Duff's Device

In 1983, while working at Lucasfilm, Tom Duff invented some C magic that implements loop unrolling without the need for the second (that is, postprocessing) loop. If you haven't seen it before, you should fasten your seatbelts before reading on:

```
int n = (len + 8 - 1) / 8;
switch (len % 8) {
    case 0: do { HAL_IO_PORT = *pSource++;
    case 7:     HAL_IO_PORT = *pSource++;
    case 6:     HAL_IO_PORT = *pSource++;
    case 5:     HAL_IO_PORT = *pSource++;
    case 4:     HAL_IO_PORT = *pSource++;
    case 3:     HAL_IO_PORT = *pSource++;
    case 2:     HAL_IO_PORT = *pSource++;
    case 1:     HAL_IO_PORT = *pSource++;
    } while (--n > 0);
}
```

If your first reaction is "But this is not legal C code!", you are not alone. But rest assured, this is legal C/C++. For background information on how this piece of code came about and an historic e-mail by Tom Duff that he wrote to Dennis Ritchie, see http://www.lysator.liu.se/c/duffs-device.html.

Just as before, the device works by unrolling the loop eight times; that is, the copy operation is executed eight times in a row. The only exception is the first time through the *do-while* loop, where fewer copy operations may be executed, as there is no guarantee that the total number of copy operations (*len*) is evenly divisible by eight. The postprocessing loop has been effectively turned into an inlined preprocessing loop.

As in the previously shown manual unrolling scheme, the loop overhead is effectively reduced to one-eighth (there is only one loop counter decrement and only one comparison against zero for every eight copy operations). If you think one-eighth is not enough, go ahead and increase the block size until you're happy, but make sure that you increase the number of cases in the switch statement accordingly.

A Reusable Duff Device

Tom Duff's Device is highly useful, technically interesting, but unfortunately, not a pleasant sight to see. This is probably the reason why my mind refuses to memorize it. Therefore, I decided to wrap the nitty-gritty details within a macro:

```
#define DUFF_DEVICE_8(aCount, aAction)  \
    do {                                \
        int count_ = (aCount);          \
        int times_ = (count_ + 7) >> 3; \
        switch (count_ & 7){            \
        case 0: do { aAction;           \
        case 7:     aAction;            \
        case 6:     aAction;            \
        case 5:     aAction;            \
        case 4:     aAction;            \
        case 3:     aAction;            \
        case 2:     aAction;            \
        case 1:     aAction;            \
            } while (--times_ > 0);     \
        }                               \
    } while (0)
```

Compared to the original device, I've changed a couple of minor things. First, I put the whole device in a dummy *do-while* block to get a local namespace to avoid variable name clashes and to enforce a trailing semicolon. Second, I've replaced the multiplication and modulo operations with right-shift and bit-wise AND operations. Even though most contemporary compilers would apply this optimization themselves, it certainly doesn't hurt to be explicit. Third, I've introduced another local variable, *count_*, to cater for cases where coders pass in complicated (expensive) expressions or function calls.

With a macro like this, the transmitter loop is reduced to a single line of code:

```
DUFF_DEVICE_8(len, HAL_IO_PORT =
                        *pSource++);
```

By the way, did you notice that there is a subtle difference between the original *for* loop and the Duff Device version? Answer: The Duff Device does the wrong thing in case *len* is zero, because it executes the operation eight — not zero — times. If you want, you can add support for this special case by adding an *if* check to the macro; however, I prefer to handle it outside the macro and add the check only if it is really needed:

```
if (len > 0)
    DUFF_DEVICE_8
        (len, HAL_IO_PORT = *pSource++);
```

As another example, consider this simplified version of an EEPROM driver routine that I recently implemented:

```
int CopyToEEPROM(const void*
      pSource, void* pDest, unsigned len) {

    // Anything to copy?
    if (len > 0) {
        const char* s = (const char*)pSource;
        char* d = (char*)pDest;

        // First, need to copy data to page buffer
        DUFF_DEVICE_8(len, *d++ = *s++);

        // Start EEPROM write cycle
        HAL_EEPROM_WRITE();

        // Check if data was transferred correctly
        s = (const char*)pSource;
        d = (char*)pDest;
        DUFF_DEVICE_8(len,
          if (*d++ != *s++) return 1; /* Fail */);
    }
    return 0; // Success
}
```

There used to be two slow, hand-coded loops in the original driver code: One copied the data to the EEPROM page buffer (as required by the EEPROM hardware), the second checked whether the data was successfully written. By replacing them with two Duff Devices, the code has become both faster and clearer.

DDJ