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++;
}
```

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++;
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++;
}
```

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:

```
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:

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

“Tom Duff invented a special kind of loop-unrolling mechanism known as ‘Duff’s Device’”

---

Ralf Holly

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

http://www.ddj.com  Dr. Dobb’s Journal, August 2005
Think, Innovate and Work with Technology Leaders

At Intel’s Enterprise Platforms group, we are focused on delivering leading technology, silicon, software and platforms that enable superior enterprise computing solutions. Enterprise computing systems and servers powered by Intel® processors can be found in small, mid-tier and high-performance environments. Our products include server and work-station components such as pro-cessors, motherboards, chipsets, server adapters, chassis, platforms and raid controllers. We currently have an opportunity for:

Server BIOS Engineers

In this position, you will develop BIOS and Extensible Firmware Interface (EFI) code, resolve customer issues, train/support offshore manufacturers and apply software engineering principles to improve quality.

This position requires a B.S. degree or a M.S. degree in Electrical Engineering or Computer Science (or equivalent work experience) with more than three years of direct experience. Additional qualifications include: applicable training and experience in the development of low-level firmware or software, preferably PC BIOS in assembler and C; working understanding of most key PC server specifications such as Intelligent Platform Management Interface (IPMI), Advanced Configuration and Power Interface (ACPI), SMBIOS*, EFI and Peripheral Component Interconnect (PCI); a strong history of resolving issues in unfamiliar code is highly desirable and a working familiarity with software quality practices.

For immediate consideration, please email your resume to epsdjobs@intel.com. Or for more information on Intel visit our website at www.intel.com/jobs.

Make the most of your mind

Think, Innovate and Work with Technology Leaders

At Intel’s Enterprise Platforms group, we are focused on delivering leading technology, silicon, software and platforms that enable superior enterprise computing solutions. Enterprise computing systems and servers powered by Intel® processors can be found in small, mid-tier and high-performance environments. Our products include server and work-station components such as pro-cessors, motherboards, chipsets, server adapters, chassis, platforms and raid controllers. We currently have an opportunity for:

Server BIOS Engineers

In this position, you will develop BIOS and Extensible Firmware Interface (EFI) code, resolve customer issues, train/support offshore manufacturers and apply software engineering principles to improve quality.

This position requires a B.S. degree or a M.S. degree in Electrical Engineering or Computer Science (or equivalent work experience) with more than three years of direct experience. Additional qualifications include: applicable training and experience in the development of low-level firmware or software, preferably PC BIOS in assembler and C; working understanding of most key PC server specifications such as Intelligent Platform Management Interface (IPMI), Advanced Configuration and Power Interface (ACPI), SMBIOS*, EFI and Peripheral Component Interconnect (PCI); a strong history of resolving issues in unfamiliar code is highly desirable and a working familiarity with software quality practices.

For immediate consideration, please email your resume to epsdjobs@intel.com. Or for more information on Intel visit our website at www.intel.com/jobs.

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

if (len > 0) {
  DUFF_DEVICE_8(len, #pSource, #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
    if (#d++ = #s++ return 1; /* Fail */)
    return 0; // Success
  }
}

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;
    if (#d++ = #s++ return 1; /* Fail */)
    return 0; // Success
  }